个性化你的内核

编程狂人

Programming Madman

NO.21

推酷

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法，从海量文章资讯中挖掘出高质量的内容，并通过分析用户的阅读偏好，准实时推荐给你最感兴趣的内容。我们推荐的内容包含科技、创业、设计、技术 、营销等内容，满足你日常的专业阅读需要。我们针对IT人还做了个活动频道，它聚合了IT圈最新最全的线上线下活动，使IT人能更方便地找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章，并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布，敬请关注阅读。

为了迎接4月23日的"世界读书日"，《编程狂人》周刊推出下载周刊送书活动。我们从 http://weibo.com/chinapub2011：@互动出版网china-pub 精选了http://weibo.com/turingbooks @图灵教育 出版的5本好书：《深入网站开发和运维》《大话重构》《iOS开发指南》《Software Design中文版01》《面向模式的软件架构》。详情请参见我们的官方微博：http://weibo.com/tuicool2012 @推酷网 活动截止日期为4月25日。

本期为精简版 周刊完整版链接：
http://www.tuicool.com/mags/53548155d91b144bdb02a865

欢迎下载推酷客户端
体验更多阅读乐趣

版权说明

本刊只用于行业间学习交流
署名文章及插图版权归原作者享有

目录

OpenStack大数据项目Sahara概述

2013年4月，OpenStack社区知名厂商Mirantis正式宣布了基于OpenStack的开源BDaaS（BigData-as-a-Service）项目——Sahara（原名Savanna），正式开始了在OpenStack上构建大数据服务能力的努力。

近日，开源技术专家章宇（@一棹凌烟）在其博客上分享了对Sahara项目的研究心得。整个介绍系列分为7篇文章，除前言部分外，其余六篇分别是：

- Sahara概述：介绍项目的目的、概况、发展等基本情况
- Sahara使用方式：介绍具体如何使用Sahara进行大数据业务操作
- Sahara设计实现：介绍Sahara的架构设计与实现
- Sahara与AWS EMR和Serengeti的对比：将Sahara与目前最知名的公有云大数据服务和Hadoop虚拟化项目进行简单对比分析
- 对Sahara的若干思考
- 小结与展望

在《Sahara概述》中，章宇介绍了Sahara的定位、功能的演进、社区支持力度与整体发展的趋势。

Sahara最初的基本定位是基于OpenStack提供简单的Hadoop集群创建方式，不过随着项目不断演进，Sahara所涵盖的范畴也有所扩大。章宇从两个层面介绍了Sahara项目的发展方向：

从服务层次的维度看，Sahara已经开始从利用OpenStack的IaaS能力，提供简单的大数据工具集群创建和管理服务，扩展到提供分析即服 务（Analytic-as-a-Service）层面的大数据业务应用能力。Sahara v0.3中引入的EDP（Elastic Data Processing）就是一个明确的体现。

从承载的业务类型维度看，Sahara也很有可能会迅速突破单一的Hadoop工具范畴，拓展支持其他新兴的大数据工具。例如，关于提供Spark支持的BP已经被提交至社区，目前正在等待review。

Sahara项目的发展较快，其项目PTL Sergey Lukjanov已经宣布Sahara将于OpenStack Juno版本中正式成为integrated项目，目前代码已经提交，并在等待review，其版本演进可以参见其wiki页面介绍。目前Sahara已经被集成在RDO中，因此可以被更为简单方便的安装部署。

《Sahara使用方式》简单介绍了Sahara的使用模式、基本概念与操作流程。

Sahara有两种使用模式：

- 基本的大数据集群应用模式（基本模式）
- 通过EDP机制引入的分析即服务模式（EDP模式）

简单来说，基本模式要求用户自己从底层搭建Hadoop虚拟机、建立集群，技术门槛较高；而EDP模式有点类似于AWS EMR服务，对底层的Hadoop集群操作和Hadoop业务操作进行了封装，暴露给用户的只有非常简单的接口，使用简便。

章宇介绍了Sahara当中的节点（node）、节点组(node group)、节点组模板（node group template）、集群（cluster）、集群模板（cluster template）、任务（job）等关键概念，并简单列出了在基本模式下用Sahara建立Hadoop集群的操作流程。整个介绍比较概括，step by step的操作文档可参考Sahara官方的QuickStart。

接下来，章宇开始从研究代码的层面介绍Sahara的设计与实现。Sahara的设计有两大特点：

1、模块化、可配置：

Sahara中的大量功能和机制，都基于可选择、可配置的模块化插件实现，例如：可以通过对engine的配置来选择不同的Hadoop集群编配引擎，通过对plugin的配置来选择不同的Hadoop发行版安装与部署方式和工具，等等。

2、代码重用：

Sahara也尽可能重用了OpenStack自身提供的IaaS层组件及其服务，例如：利用Nova提供虚拟机资源，利用Horizon提供人机界面，等等。

Sahara对Horizon（界面）、Glance（镜像管理）、Keystone（身份认证）、Heat（集群配置）、 Ceilometer（监控）、Nova（虚拟机管理）、Neutron（网络）、Cinder（块存储）和Swift（对象存储）都有不同程度的代码复 用，其中Nova、Glance和Keystone是必要组件，其他组件可选用。

Sahara的整体架构可参考其架构图。其中，章宇建议：

在分析集群创建流程时，主要应关注sahara.api、sahara.service.api、sahara.service.engine和 sahara.plugins这四个package的各自行为及相互关系。其中，sahara.service.api中的 _provision_cluster()驱动了整个cluster创建的过程。

接下来，章宇从产品和技术的层面将Sahara与EMR、Serengeti进行了对比，要点如下：

- EMR在Sahara基本模式的基础上融合了EDP模式的特点
- EDP的用户只需要指定“哪些数据”、“哪个集群”、“哪个程序包”这三要素，而完全不用关心集群如何创建、如何管理这样的与自己核心业务诉求无关的问题
- EMR的用户则除了需要在创建集群时指定大量信息外，还需要负责集群和业务的运行管理
- 比较而言，EDP的用户是纯粹的大数据业务应用者，而EMR的用户则身兼业务应用和系统运维两种职责
- 基于EMR的大数据解决方案，全面涵盖了数据的存储、计算、分析、共享等各个处理环节，这是Sahara还难以企及的
- Sahara和Serengeti的区别，可以说是“应用云化”和“应用虚拟化”的区别。Serengeti项目的主要关注点在于如何为搭建在虚拟机环境下的Hadoop集群提高性能和可靠性，这里面的思考是Sahara可以借鉴的

介绍到这里，章宇对Sahara目前的状态进行了概述，认为目前的Sahara还面临以下几点挑战：

- Sahara的管理平面性能存在疑问，创建和发布集群的等待时间有待测试
- 复杂管理的成功率方面，目前Sahara中没有看到明确的处理机制，这是一个缺失

- Sahara搭建的Hadoop在虚拟化环境下的性能有待优化，不过这个问题可以等到前面两个关键问题解决了之后再来优化
- Auto-scaling的缺失。目前Sahara要扩展需要人工执行
- Sahara最大的亮点在EDP，其价值有待进一步挖掘

原文作者：杨赛
原文链接：http://www.infoq.com/cn/news/2014/04/openstack-sahara

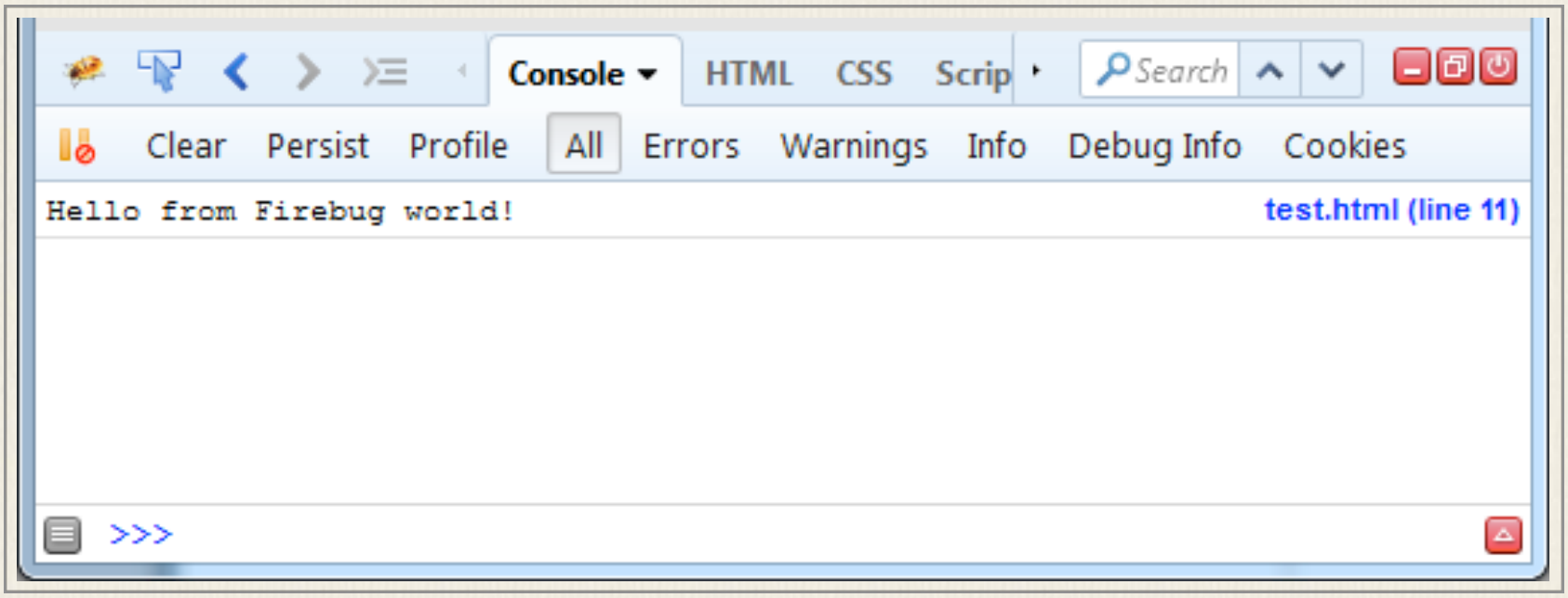
前端神器 Firebug 2.0 新特性一览

如果你从事Web前端方面的开发工作，那么对Firebug一定不会陌生，这是Firefox浏览器的一款插件，集HTML查看和编辑、Javascript控制台、网络状况监视器于一体，给Web开发者带来了极大的便利，堪称Web前端开发神器。

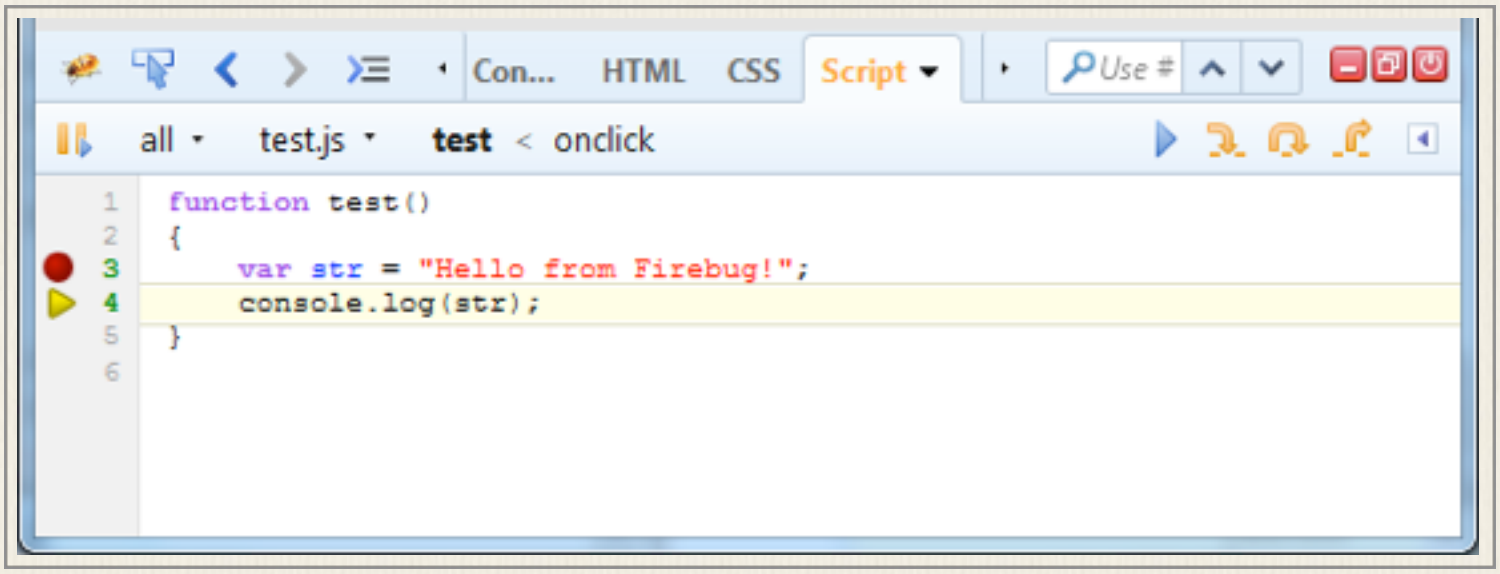
最新版本Firebug 2.0即将发布，下面就来看看这个大版本中有哪些改进。

1. 基于新的Firefox调试引擎（JSD2）

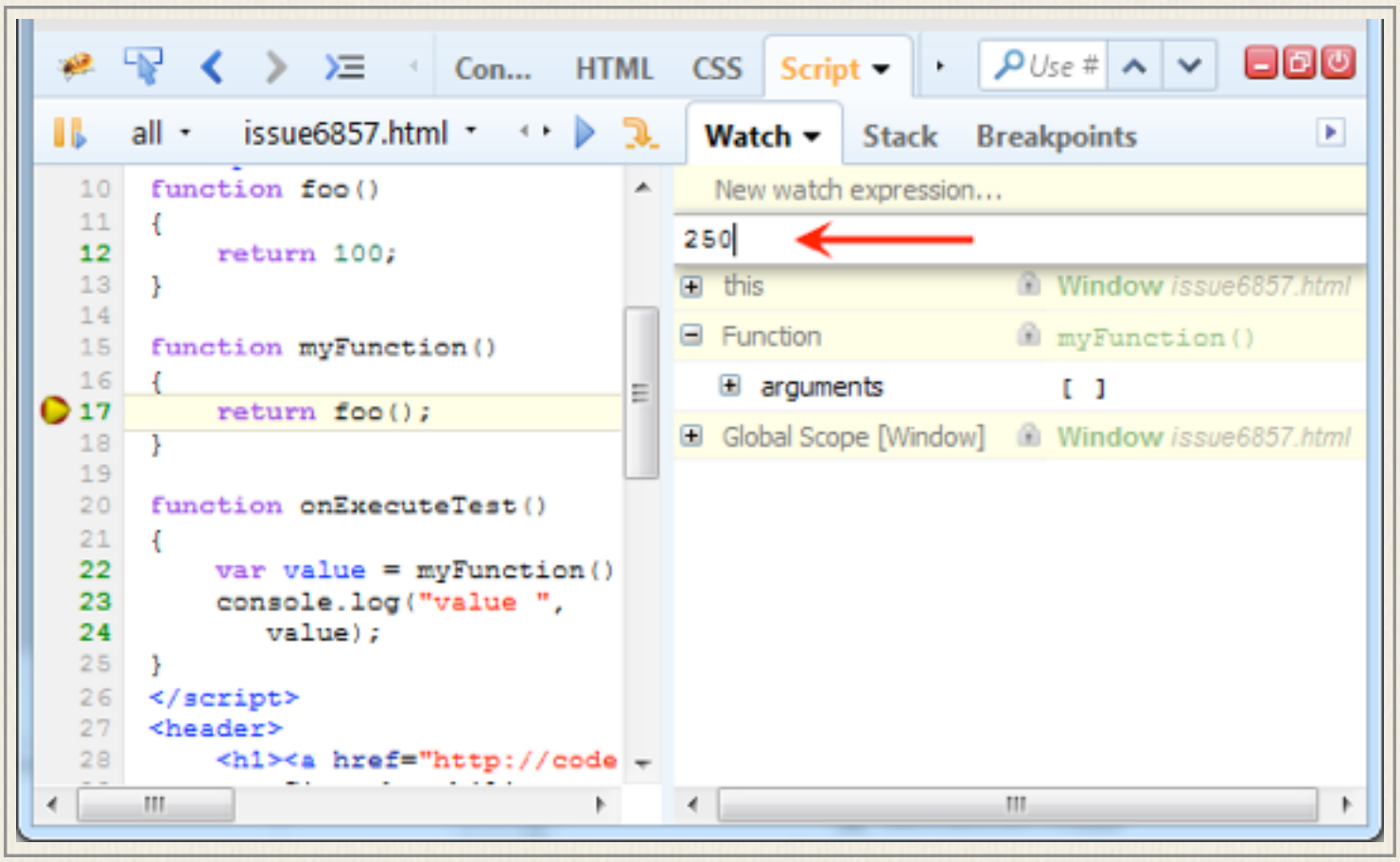
2. 新的UI，以匹配Firefox 29版本中引入的Australis主题



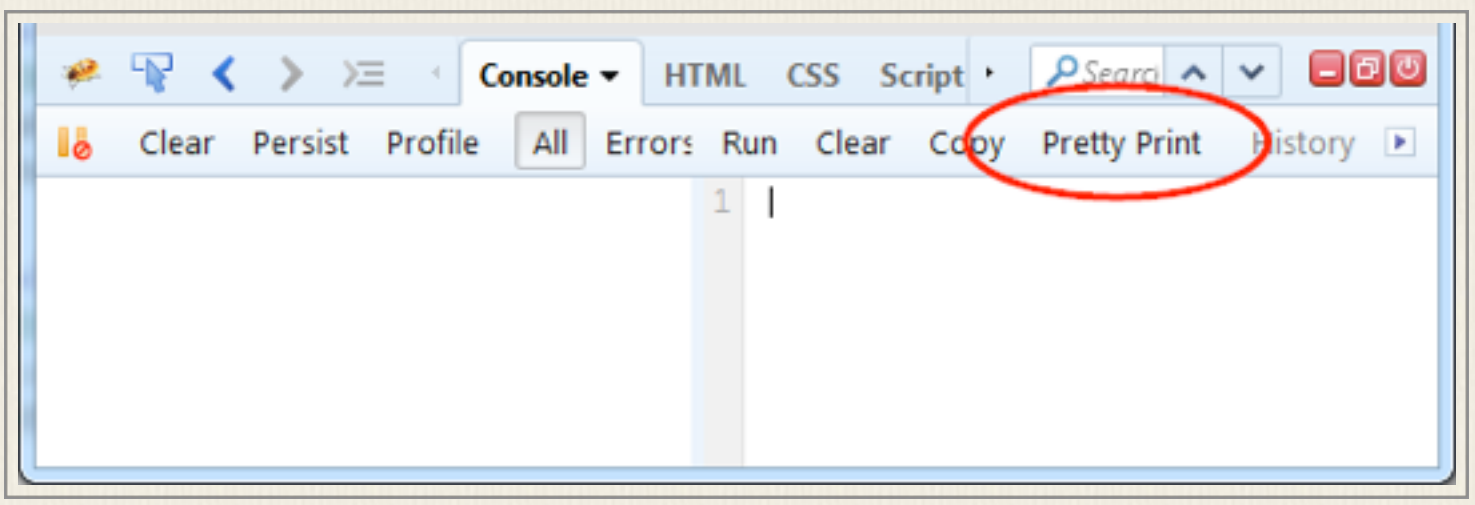
3. Script面板支持语法高亮



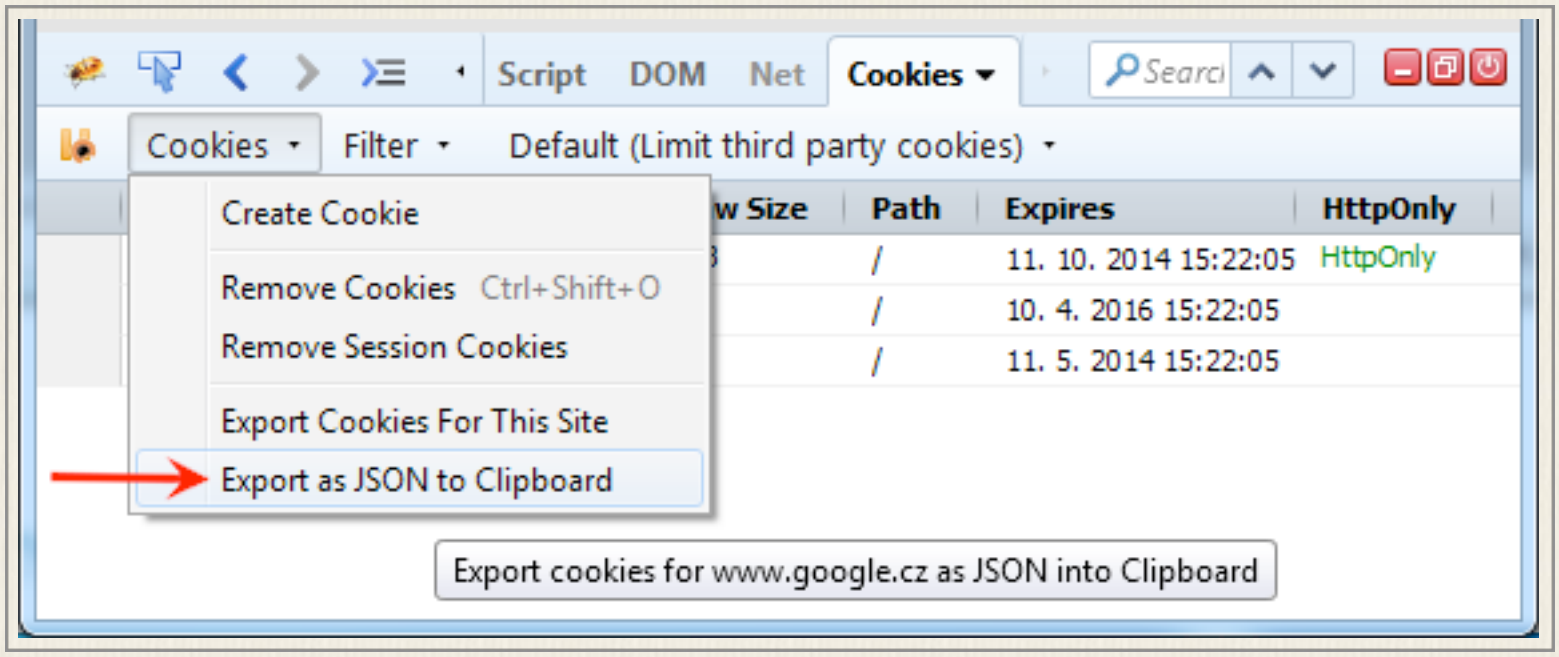
4. 允许检查和修改JavaScript函数的返回值



5. Console面板和Script面板新增了Pretty Print（源码美化输出）功能



6. 新增了Export as JSON to Clip-board功能，允许导出当前页面中所有的cookies到剪贴板中。



更多信息：https://blog.getfirebug.com/

最新版下载地址：firebug-2.0b1.xpi（支持Firefox 30+版本）

Firefox v30下载（Aurora版）：http://www.mozilla.org/en-US/firefox/aurora/

Firefox v31下载（Nightly版）：http://nightly.mozilla.org/

作者：wangguo
原文链接：http://www.iteye.com/news/28960

用CSS3和JavaScript开发《街头霸王》游戏

译者/santiago

微博：@歪脖骇客

网址：http://www.webhek.com/

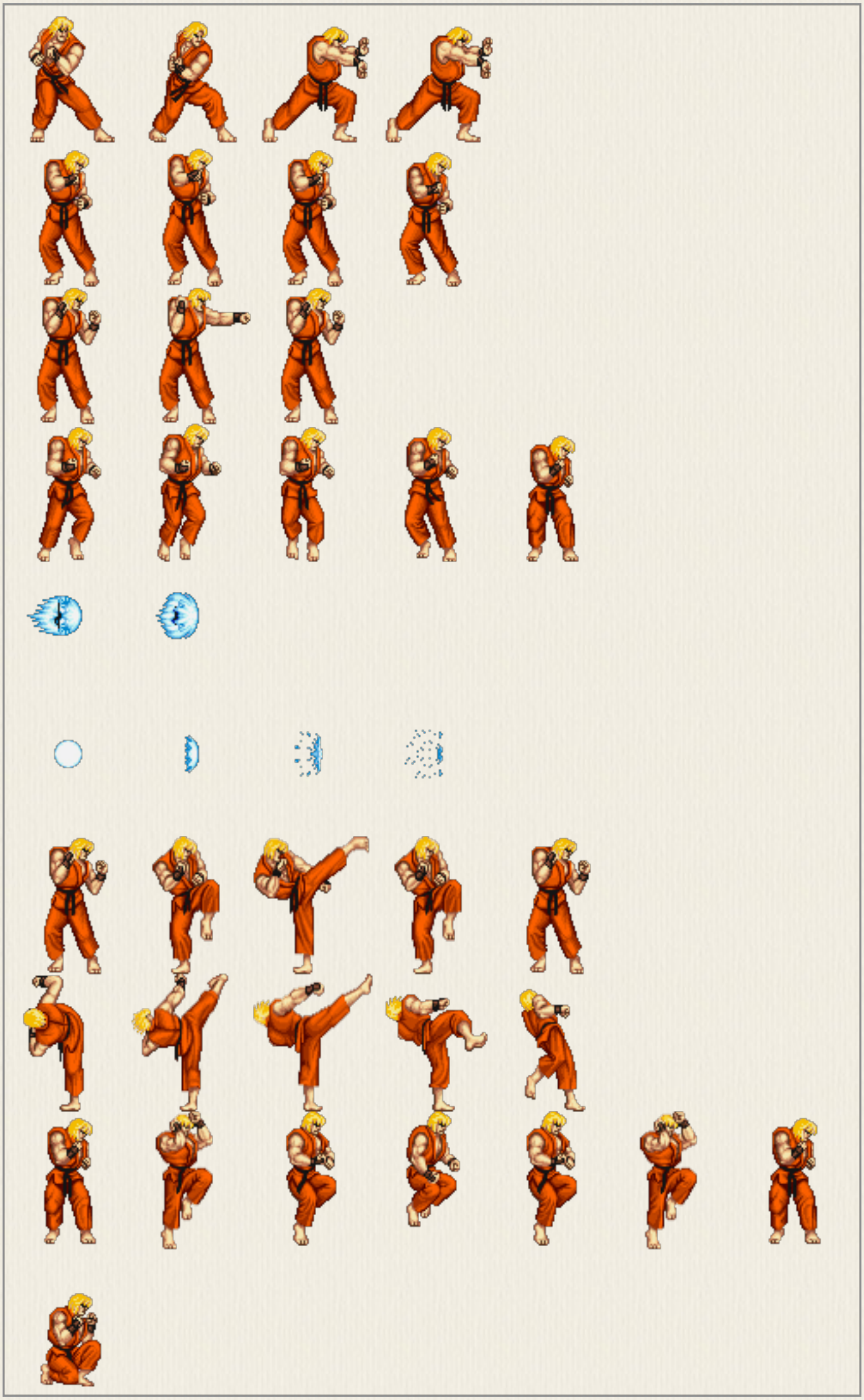
最近我学到了一种很酷的技术，使用CSS3的steps()动画属性来生成PNG背景动画。这种技术的主要功能是利用铺贴PNG背景图的方式“重现”GIF图的动画效果。

我的朋友都知道，年轻时候我非常痴迷《街头霸王》游戏，而当我看到了这个素材....你知道我的脑子里想到了什么吗?

观看演示

让我们来做出第一个CSS动作

首先我们要实现的是直拳(就是下面图案中的第三个)。第一步是打开Photoshop，抠出这几个图，这些图都要保持着70px宽和80px高。有一个很好的网站Texture Packer，它能帮你处理这些图片。最终，你的图片应该是这个样子:



下面我们需要有一个DIV，动作将显示到这个DIV里：

```
/* html */
<div class="ken lazy "></div>
/* css */
.ken {
    width:70px; height:80px; /* exactly the size of an image in our
sprite */
    background-image:url('../images/sprite.png');
}
```

这里我省略了可能需要的浏览器CSS前缀。那么，打直拳的CSS代码就应该是这样：

```
/* css */
.punch {
    animation: punch steps(4) 0.15s infinite;
}
@keyframes punch {
    from { background-position:0px -160px; }
    to { background-position:-280px -160px; }
}
```

我们这里做的就是定义一个动画CSS类(.punch)，基本的动作就是让background-position从0px运动到-280px(沿x轴)。这个动作有四帧(steps(4)就是对应的4副打拳的图片)，这个动作在0.15秒里完成；这里的设置是循环播放。

我们还需要有个增加/删除DIV.ken上的.punch类的方法，当控制键被按下时，会调用这个方法。

```
/* javascript */
$(document).on('keydown', function(e) {
    if (e.keyCode === 68) { // 68 是键盘上的D字符
        $('.ken').addClass('punch');
        setTimeout(function() {
           $ken.removeClass('punch'); }, 150);
    }
});
```

如果有人按下了“D”键，我们使用jQuery的addClass('punch')来增添CSS类，而且用setTimeout设置了150毫秒时间的延迟后，删除它(因为我们的动作是在0.15s完成的)。基本上这就是创造所有动画需要的基础知识了。

使用SASS改进代码

如果你留心我们所做的事情，你会发现代码里有些值永远不会改变(图片的长宽等)，而且，在创建了一些动画后，你会发现有很多重复的代码，这会让我们的代码很难阅读和维护。SASS能帮助我们消除重复的代码。

首先我们要创建几个基本的@mixins，比如animation()和keyframes()：

```
@mixin animation($params) {
    -webkit-animation:$params;
    -moz-animation:$params;
    -ms-animation:$params;
    animation:$params;
}
@mixin keyframes($name) {
    @-webkit-keyframes $name { @content }
    @-moz-keyframes    $name { @content }
    @-ms-keyframes     $name { @content }
    @keyframes         $name { @content }
}
```

我们要把图片的长度和宽度存储到变量里，SASS变量在这里就有了很大的用处：

```
$spriteWidth:70px;
$spriteHeight:80px;
```

最终，我们把这些组合到一起，创建一个复杂的SASS mix-in，用来声明我们的动画动作和计算背景移动的位置：

```
@mixin anim($animName, $steps, $animNbr, $animParams){
    .#{$animName} {
        @content;
        @include animation($animName steps($steps)
                    $animParams);
    }
    @include keyframes($animName) {
        from { background-position:0px (-$spriteHeight
               * ($animNbr - 1)); }
        to { background-position:-($spriteWidth *
           $steps) (-$spriteHeight * ($animNbr - 1)); }
```

```
    }
}
```

现在，你可以用一行代码创建一个动画动作：

```
$spriteWidth:70px;
$spriteHeight:80px;

/* 直拳 */
@include anim($animName:punch, $steps:3, $animNbr:3, $animParams:.15s
infinite);
/* 踢腿 */
@include anim($animName:kick, $steps:5, $animNbr:7, $animParams:.5s
infinite);
/* 波動拳 */
@include anim($animName:hadoken, $steps:4, $animNbr:1,
$animParams:.5s infinite);
...
```

这个$animNbr变量很重要：我们的动画中的计算要基于这个数字。事实上，它就是动画中的动作序列的编号。我们第一个例子是长拳，它在图片中的位置是3，踢腿是7，以此类推。

对光电球发生碰撞的检测

我需要一个很高的频率对碰撞进行检测。每50毫秒对光电球的位置(偏移)检查一次，如果光电球的偏移量超出了窗口宽度，这意味着这个球要撞击到边框，我们要马上应用.explodeCSS类。

下面就是如何实现的；虽然不完美，但完全可以运行：

```
var $fireball = $('<div/>', { class:'fireball' });
$fireball.appendTo($ken);

var isFireballColision = function(){
    return $fireballPos.left + 75 > $(window).width();
};

var explodeIfColision = setInterval(function(){
    $fireballPos = $fireball.offset();
    if (isFireballColision()) {
        $fireball.addClass('explode');
```

```
            clearInterval(explodeIfColision);
            setTimeout(function() { $fireball.remove(); }, 500);
        }
    }, 50);
```

观看演示

下一步需要做的

我们可以很容易的往里面添加一些声效、背景音乐、其它视觉效果，还可以增加web RTC功能，让多台计算机控制多个人物。(也许你可以使用NodeJS和Socket.io，或Meteor frame-work)；这些就是我喜欢web开发的原因：给你无限可能。

(英文：Build a Street Fighter Demo with CSS Animations and JavaScript.)

原文链接：http://www.webhek.com/street-fighter/

使用 Google Analytics 跟踪 JavaScript 错误

作者：山边小溪
微博：@山边小溪
网址：jizhula.com

Google Analytics（谷歌分析）不仅仅是一个流量统计工具，你还可以用它来测量广告活动的有效性，跟踪用户多远到所需的页面流（从点击广告到购物车到结账页面）获取，并基于用户的信息设置浏览器和语言环境支持。

但是，这一切东西都不是我们开发者所需要的。今天给大家介绍一个很另类的技巧，通过自定义事件跟踪 JavaScript 错误。下面就让我们来看看如何实现的错误检查的：

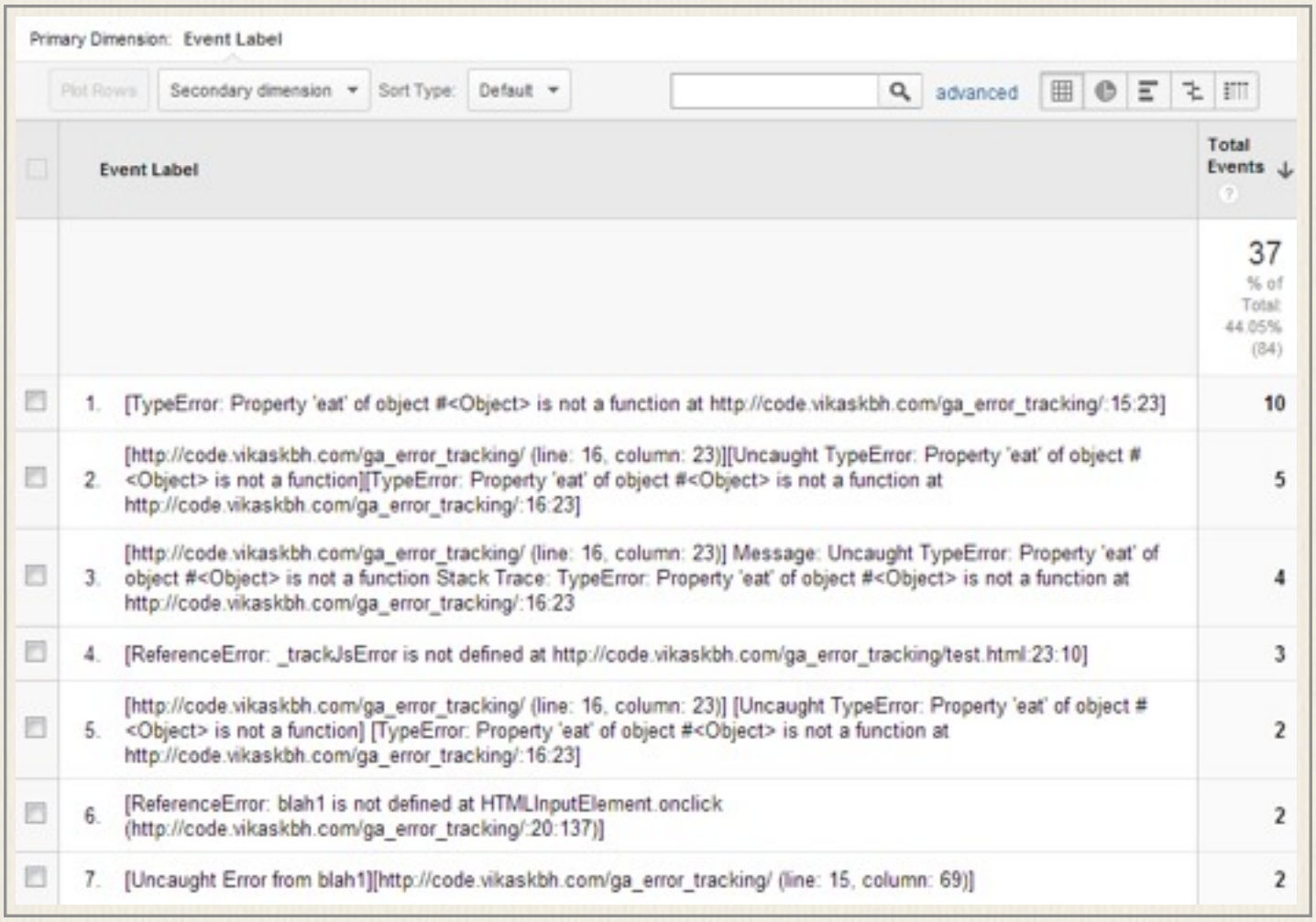
```
// 跟踪基本的 JavaScript 错误信息
window.addEventListener('error', function(e) {
    _gaq.push([
        '_trackEvent',
        'JavaScript Error',
        e.message,
        e.filename + ':  ' + e.lineno,
        true
    ]);
});

// 跟踪 AJAX 错误信息 (jQuery API)
$(document).ajaxError(function(e, request, settings) {
    _gaq.push([
        '_trackEvent',
        'Ajax error',
        settings.url,
        e.result,
        true
    ]);
});
```

现在，当你进入谷歌分析，您可以在查看其他网站统计的同时看到自定义事件的信息。当然，你会告诉营销人员这些都不是真正的错误，那是功能（偷笑），那是另一个故事了。

查看错误报告

要有这个功能，先确保已有 Google Analytics 账号，查看步骤：

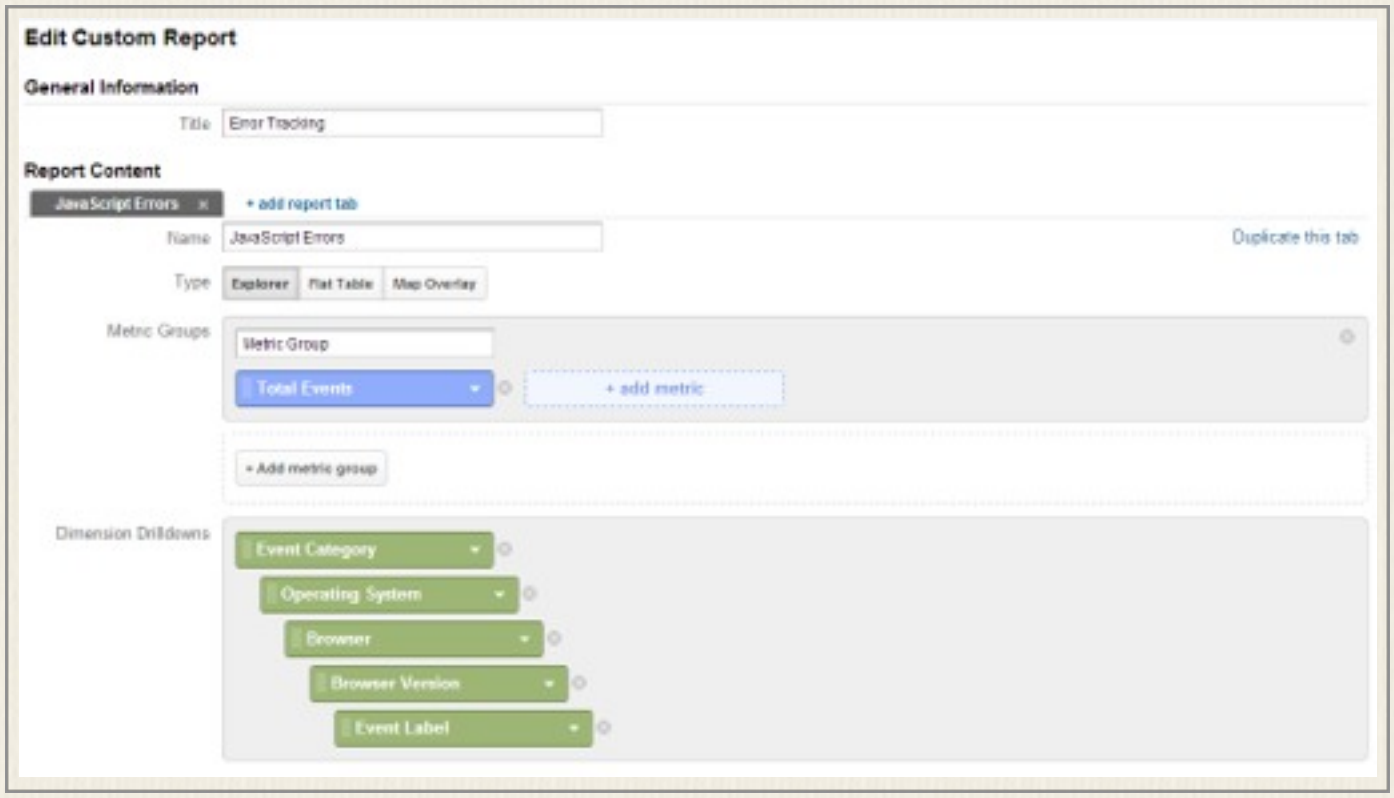
- 登录 Google Analytics；
- 在左侧的导航面板找到行为；
- 点击事件查看概览；
- 在事件分类下面会有异常，点击；
- 在出现的界面点击事件标签，点击维度过滤；
- 这时候就能看到跟踪到的所有错误信息了；



自定义错误报告

Google Analytics 提供了强大的报告自定义功能，你可以自定义错误报告来跟踪产生错误的操作系统，浏览器以及版本等信息。设置步骤：

- 找到自定义功能，在报告选项卡的旁边；
- 点击新增自定义报告；
- 生成信息 = 错误跟踪；
- 报告内容 => 名称 = JavaScript 错误；
- Metric Group = 所有事件；
- 维度结构 => 事件分类 > 操作系统 > 浏览器 > 浏览器版本 > 事件标签；
- 点击保存；



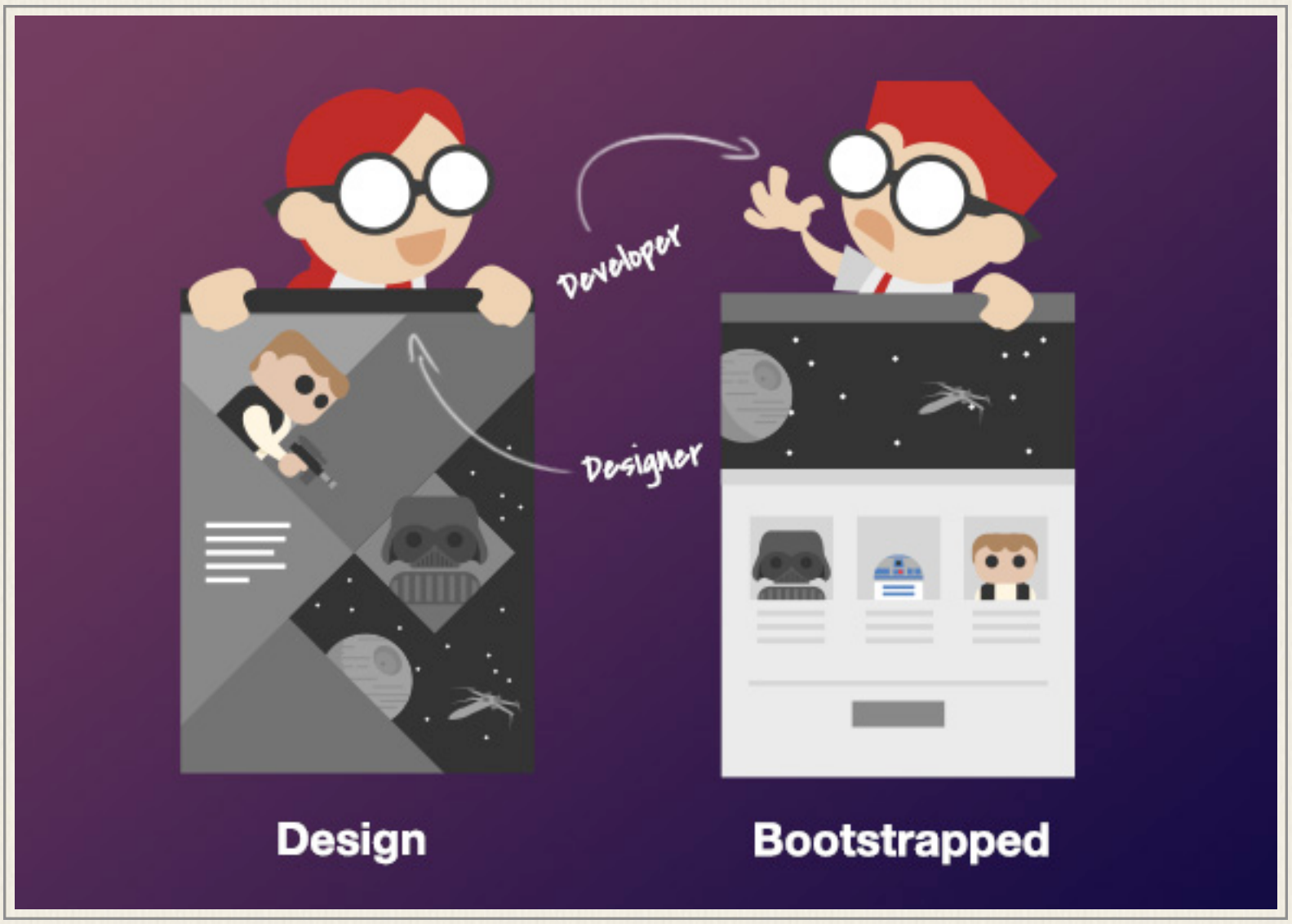
原文链接：http://www.cnblogs.com/lhb25/p/track-errors-google-analytics.html

5 个不用 Bootstrap 的理由

译者：地狱星星, 赵亮-碧海情天, andylam

在以前我们的博客文章中，我们讨论了在web设计和开发项目中使用Twitter Bootstrap的好处。

Twitter Bootstrap也有很多的缺点。让我们看看这些主要的问题：



1，它不遵循最佳实践

我们在使用Twitter Bootstrap时遇到的最大问题之一是你的DOM元素上将拥挤大量的类。这打破了良好的web设计基本规则之一，HTML不再有语义，而且内容和表示 不再分离。前端纯粹主义者会觉得这相当令人讨厌，以为它使可扩展性、重用性和维护性遇到了更大的挑战。表示和交互不再独立于内容在Twitter Bootstrap中也被进一步的强化。

```
<hr class="featurette-divider">
▼<div class="featurette">
  <img class="featurette-image pull-right" src="../assets/img/examples/browser-icon-chrome.png">
 ▶<h2 class="featurette-heading">…</h2>
 ▶<p class="lead">…</p>
</div>
<hr class="featurette-divider">
▼<div class="featurette">
  <img class="featurette-image pull-left" src="../assets/img/examples/browser-icon-firefox.png">
 ▶<h2 class="featurette-heading">…</h2>
 ▶<p class="lead">…</p>
</div>
```

哦，如此多不必要的类!

2. 它将与我现有设置发生碰撞

如果你被空投到一个干了一半的大项目中，想要使用 Twitter Bootstrappy 享受其所有的好处会如何呢？糟糕的是，你会碰到一大堆的问题，冲突首先会从生成 HTML、CSS和JavaScript 开始。然后是它们的资源，你必须深入项目中那些阴暗的角落，搞清楚哪些脚本和样式需要删除或替换。Twitter Bootstrap 会潜在创建额外的工作，当你深入项目会不可避免地发现和修复奇怪的错误，你为自己辩护的理由将会击败你优先选用它的目的。

3. Twitter Bootstrap 太重

坦率的说，Twitter Bootstrap包括126kb的CSS和29kb的JavaScript。如果你想要使用Twitter Bootstap的所有功能，你应该好好考虑资源的加载时间。当然，对于一些地方这可能不是问题，但是在新西兰互联网不得不横跨太平洋，这时数据达到那儿 将是很缓慢的。因此考虑你的目标市场。Twitter Bootstrap将帮助你建立一个有吸引力的、响应式的网站，但是一些手机用户将因为缓慢的加载时间和消耗电量的脚本而别拒之门外。

4. 不支持SASS

可能是最大的争论之一，BootStrap使用Less构建，原生不支持Compass和SASS。现在请不要误会我的意思，Less是好的，我以前使用它，它肯定有它的优点。但是SASS是更好的，带有一个类似于Compass的框架，使用它好像完全不需要过多的考虑。一些人建立了Compass gem的Bootstrap，但是坦率的说，你将不得不使用Less。在将来的文章中，我将更多的讨论SASS和Less。与此同时，Chris Coyier已经写了一篇文章比较两者。

5. “喂!我的新站看起来跟大伙的一样!”

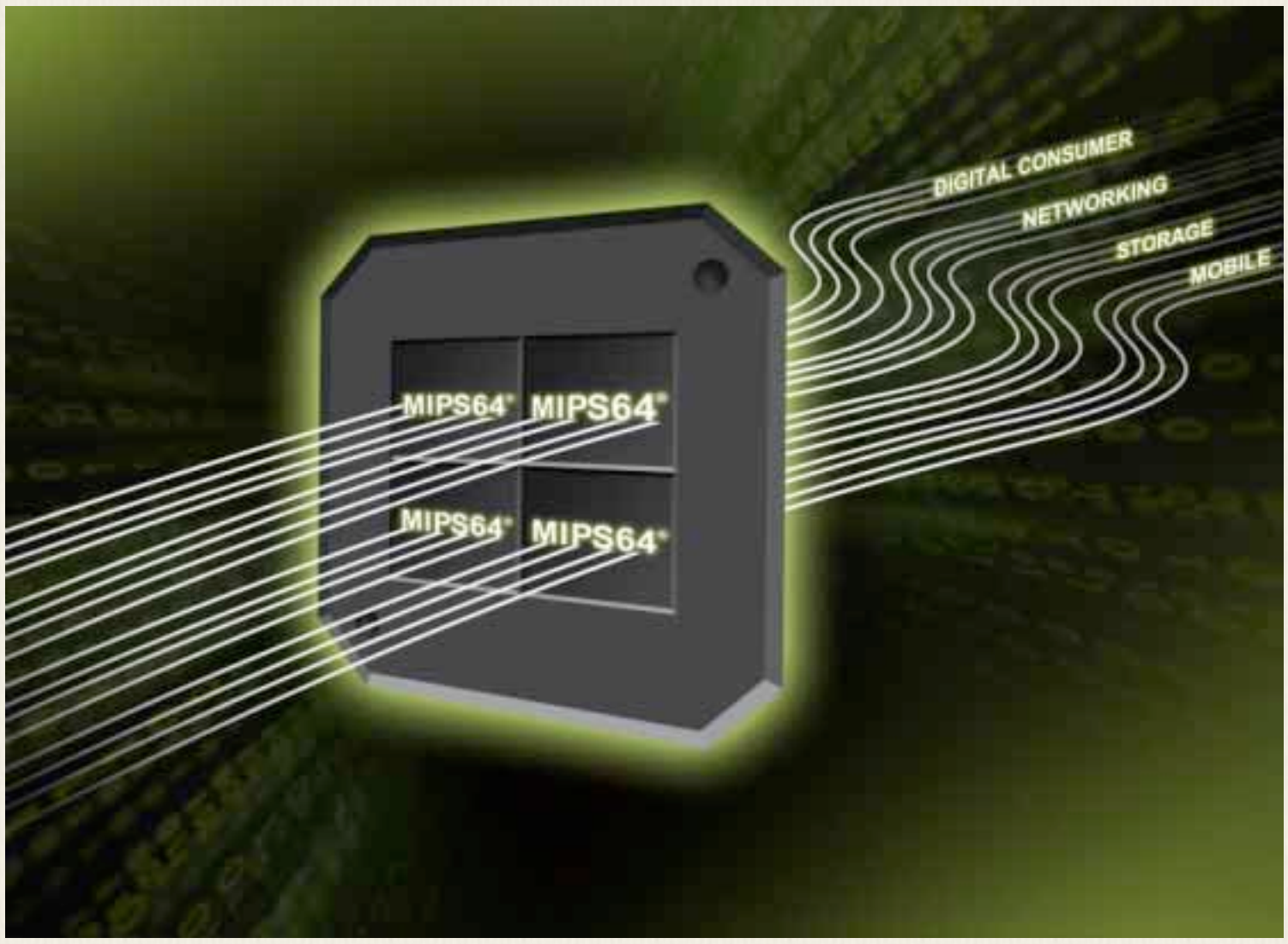
Twitter Bootstrap超级流行，流行到所有开发人员和他家的狗都去用的程度。你可能发觉由于时间限制，定制你的app或者网站时被迫使用了很多原生 Bootrasp风格。这会导致无意创建很多类似的，一般的和无眼缘的网站。在Twitter Bootstrap 实现起来既快速又容易的同时，创意往往是妥协的结果。在受限的时间里，在Bootstrap结构化的环境中，实现打破常规的创新设计是很难的。

本文链接：http://www.oschina.net/translate/5-reasons-not-to-use-twitter-bootstrap

英文链接：http://www.zingdesign.com/5-reasons-not-to-use-twitter-bootstrap/

Java中使用akka手记一

作者：@54chen
网站：54chen.com
五四陈科学院特聘院长和特聘老师和特聘厕所保洁员,旧金山,拥抱过变化,人人人,小米现役码农,职业北漂陈老师



什么是actor?

- Actor模型在并发编程中是比较常见的一种模型。很多开发语言都提供了原生的Actor模型。例如erlang,scala等。
- 它由Carl Hewitt于上世纪70年代早期提出，目的是为了解决分布式编程中一系列的编程问题。
- Actor模型的本质已经被强调了无数遍：万物皆Actor。Actor之间只有发送消息这一种通信方式。
- 一个Actor如何处理多个Actor的请求呢？它先建立一个消息队列，每次收到消息后，就放入队列，而它每次也从队列中取出消息体来处理。通

常我们都使得这个过程是循环的。让Actor可以时刻处理发送来的消息。

什么是akka?

Akka是一个用Scala编写的库，用于简化编写容错的、高可伸缩性的Java和Scala的Actor模型应用。

- 下面以在java项目中使用akka写代码为例子。

依赖

- maven项目
- java6 or 7
- 添加akka相关的包

```
<dependency>
            <groupId>com.typesafe.akka</groupId>
            <artifactId>akka-actor_2.10</artifactId>
            <version>2.3.1</version>
        </dependency>
        <dependency>
            <groupId>com.typesafe.akka</groupId>
            <artifactId>akka-remote_2.10</artifactId>
            <version>2.3.1</version>
        </dependency>
        <dependency>
            <groupId>com.google.protobuf</groupId>
            <artifactId>protobuf-java</artifactId>
            <version>2.5.0</version>
        </dependency>
        <dependency>
            <groupId>com.typesafe.akka</groupId>
            <artifactId>akka-testkit_2.10</artifactId>
            <version>2.3.1</version>
        </dependency>
```

依赖包解析

- akka-actor 核心包，有这个包就可以写简单的代码了
- akka-remote 远程包，有这个包，才能够跨进程和网络调用
- protobuf-java 不解释了，之所有是要声明版本，是因为pb的版本太低会造成消息传递过程中序列化反序列化有问题

- akka-testkit 测试集，有这个包，写test case方便

常见问题

- Q:shutting down JVM since ‘akka.jvm-exit-on-fatal-error’ is enabled
- A:所有出错的时候都会有这个提示，快速错误退出是一个常见的机制，让系统最快时间发现错误。
- Q:java.lang.ClassNotFoundException: akka.remote.RemoteActorRefProvider
- A:没有添加进来akka-remote的时候会这样
- Q:class akka.remote.WireFormats$AkkaControlMessage overrides final method getUnknownFields.()Lcom/google/protobuf/UnknownFieldSet
- A:这是pb版本太低导致的，声明到2.5.0或以上

例子

- 这是typesafe的经典例子。
- 所有actor的配置都在classpath中。
- 此例启动了两个system（简称为worker与creator）：startRemoteWorkerSystem && startRemoteCreationSystem
- worker使用calculator.conf，在2552端口侦听。
- creator使用remotecreation.conf，定义了它的worker在远程的2552端口，路径在creationActor下，自己的端口为2554。

creator中的逻辑

- creator启动后，调用了schedule，进行了一秒一次的随机调用乘法或除法。
- 具体的计算，在creationActor这个actor中完成。
- 而creationActor这个actor被定义到了远程2552端口的进程中执行。

```
akka {
  actor {
    deployment {
      "/creationActor/*" {
        remote = "akka.tcp://CalculatorWorkerSystem@127.0.0.1:2552"
      }
    }
  }
```

运行中进程观察

- run CreationApplication.java
- 一个进程 启动了一个端口
- 进程通过这个端口，产生随机算式，交给另一个进程（这里是同一个进程）。

代码

- 本文提及代码在 https://github.com/XiaoMi/rose/tree/master/rose-example

文章转载自：五四陈科学院[http://www.54chen.com]

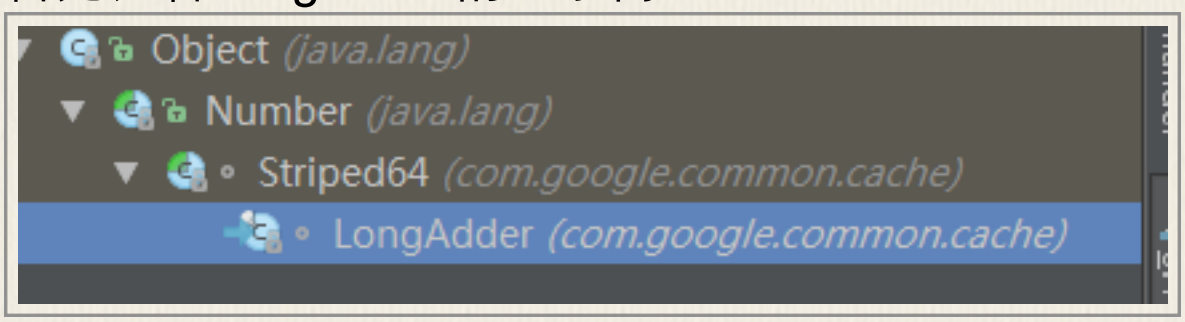
比AtomicLong还高效的LongAdder 源码解析

作者：@Jd刘锟洋
网站：www.liuinsect.com
独立博主，因上努力，果上随缘。码梦为生

接触到AtomicLong的原因是在看guava的LoadingCache相关代码时，关于LoadingCache，其实思路也非常简单清晰：用模板模式解决了缓存不命中时获取数据的逻辑，这个思路我早前也正好在项目中使用到。

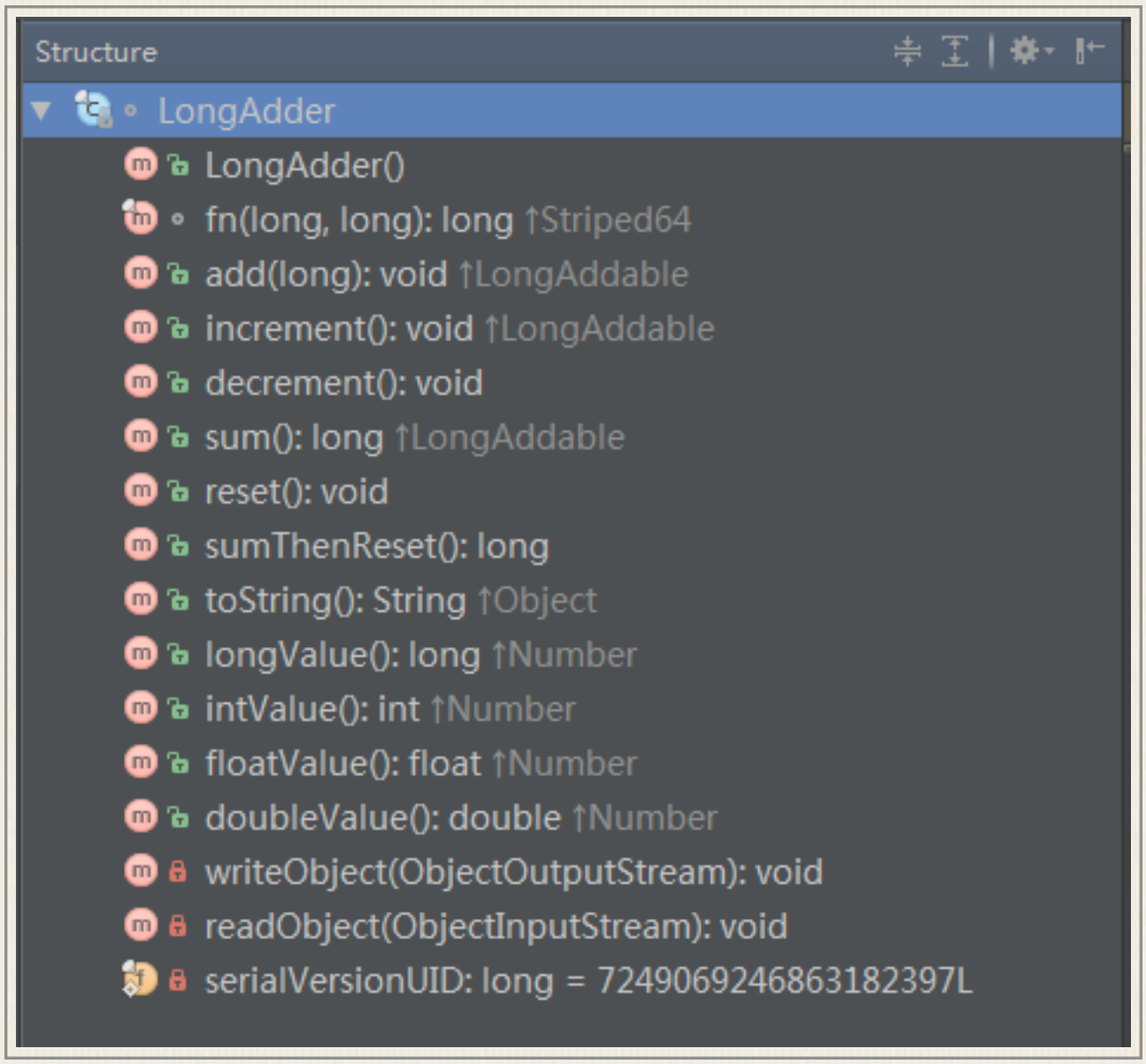
言归正传，为什么说LongAdder引起了我的注意，原因有二：
1. 作者是Doug lea ，地位实在举足轻重。
2. 他说这个比AtomicLong高效。
我们知道，AtomicLong已经是非常好的解决方案了，涉及并发的地方都是使用CAS操作，在硬件层次上去做 compare and set操作。效率非常高。

因此，我决定研究下，为什么LongAdder比AtomicLong高效。
首先，看LongAdder的继承树：



继承自Striped64，这个类包装了一些很重要的内部类和操作。稍候会看到。
正式开始前，强调下，我们知道，AtomicLong的实现方式是内部有个value变量，当多线程并发自增，自减时，均通过cas 指令从机器指令级别操作保证并发的原子性。

再看看LongAdder的方法：



怪不得可以和AtomicLong作比较，连API都这么像。我们随便挑一个API入手分析，这个API通了，其他API都大同小异，因此，我选择了add这个方法。事实上,其他API也都依赖这个方法。

```
public void add(long x) {
    Cell[] as; long b, v; HashCode hc; Cell a; int n;
    if ((as = cells) != null || !casBase(b = base, b + x)) {
        boolean uncontended = true;
        int h = (hc = threadHashCode.get()).code;
        if (as == null || (n = as.length) < 1 ||
            (a = as[(n - 1) & h]) == null ||
            !(uncontended = a.cas(v = a.value, v + x)))
            retryUpdate(x, hc, uncontended);
    }
}
```

LongAdder中包含了一个Cell 数组，Cell是Striped64的一个内部类，顾名思义，Cell 代表了一个最小单元，这个单元有什么用，稍候会说道。先看定义：

```
static final class Cell {
    volatile long p0, p1, p2, p3, p4, p5, p6;
    volatile long value;
    volatile long q0, q1, q2, q3, q4, q5, q6;
    Cell(long x) { value = x; }

    final boolean cas(long cmp, long val) {
        return UNSAFE.compareAndSwapLong(this, valueOffset, cmp, val
    }

    // Unsafe mechanics
    private static final sun.misc.Unsafe UNSAFE;
    private static final long valueOffset;
    static {
        try {
            UNSAFE = getUnsafe();
            Class<?> ak = Cell.class;
            valueOffset = UNSAFE.objectFieldOffset
                (ak.getDeclaredField("value"));
        } catch (Exception e) {
            throw new Error(e);
        }
    }

}
```

Cell内部有一个非常重要的value变量，并且提供了一个CAS更新其值的方法。
回到add方法：

```
public void add(long x) {
    Cell[] as; long b, v; HashCode hc; Cell a; int n;
    if ((as = cells) != null || !casBase(b = base, b + x)) {
        boolean uncontended = true;
        int h = (hc = threadHashCode.get()).code;
        if (as == null || (n = as.length) < 1 ||
            (a = as[(n - 1) & h]) == null ||
            !(uncontended = a.cas(v = a.value, v + x)))
            retryUpdate(x, hc, uncontended);
    }
}
```

这里，我有个疑问，AtomicLong已经使用CAS指令，非常高效了（比起各种锁），LongAdder如果还是用CAS指令更新值，怎么可能比AtomicLong高效了？ 何况内部还这么多判断！！！

这是我开始时最大的疑问，所以，我猜想，难道有比CAS指令更高效的方式出现了？ 带着这个疑问，继续。
第一if 判断，第一次调用的时候cells数组肯定为null,因此，进入cas-Base方法：

```
final boolean casBase(long cmp, long val) {
    return UNSAFE.compareAndSwapLong(this, baseOffset, cmp, val);
}
```

原子更新base没啥好说的，如果更新成功，本地调用开始返回，否则进入分支内部。
什么时候会更新失败？ 没错，并发的时候，好戏开始了，AtomicLong的处理方式是死循环尝试更新，直到成功才返回，而LongAdder则是进入这个分支。

分支内部，通过一个Threadlocal变量threadHashCode 获取一个HashCode对象，该HashCode对象依然是Striped64类的内部类，看定义：

```
static final class HashCode {
    static final Random rng = new Random();
    int code;
    HashCode() {
        int h = rng.nextInt(); // Avoid zero to allow xorShift rehash
        code = (h == 0) ? 1 : h;
    }
}
```

有个code变量，保存了一个非0的随机数随机值。

回到add方法：

```
public void add(long x) {
    Cell[] as; long b, v; HashCode hc; Cell a; int n;
    if ((as = cells) != null || !casBase(b = base, b + x)) {
        boolean uncontended = true;
        int h = (hc = threadHashCode.get()).code;
        if (as == null || (n = as.length) < 1 ||
            (a = as[(n - 1) & h]) == null ||
            !(uncontended = a.cas(v = a.value, v + x)))
            retryUpdate(x, hc, uncontended);
    }
}
```

拿到该线程相关的HashCode对象后，获取它的code变量，as[(n-1)h] 这句话相当于对h取模，只不过比起取摸，因为是 与 的运算所以效率更高。

计算出一个在Cells 数组中当先线程的HashCode对应的 索引位置，并将该位置的Cell 对象拿出来更新cas 更新它的value值。

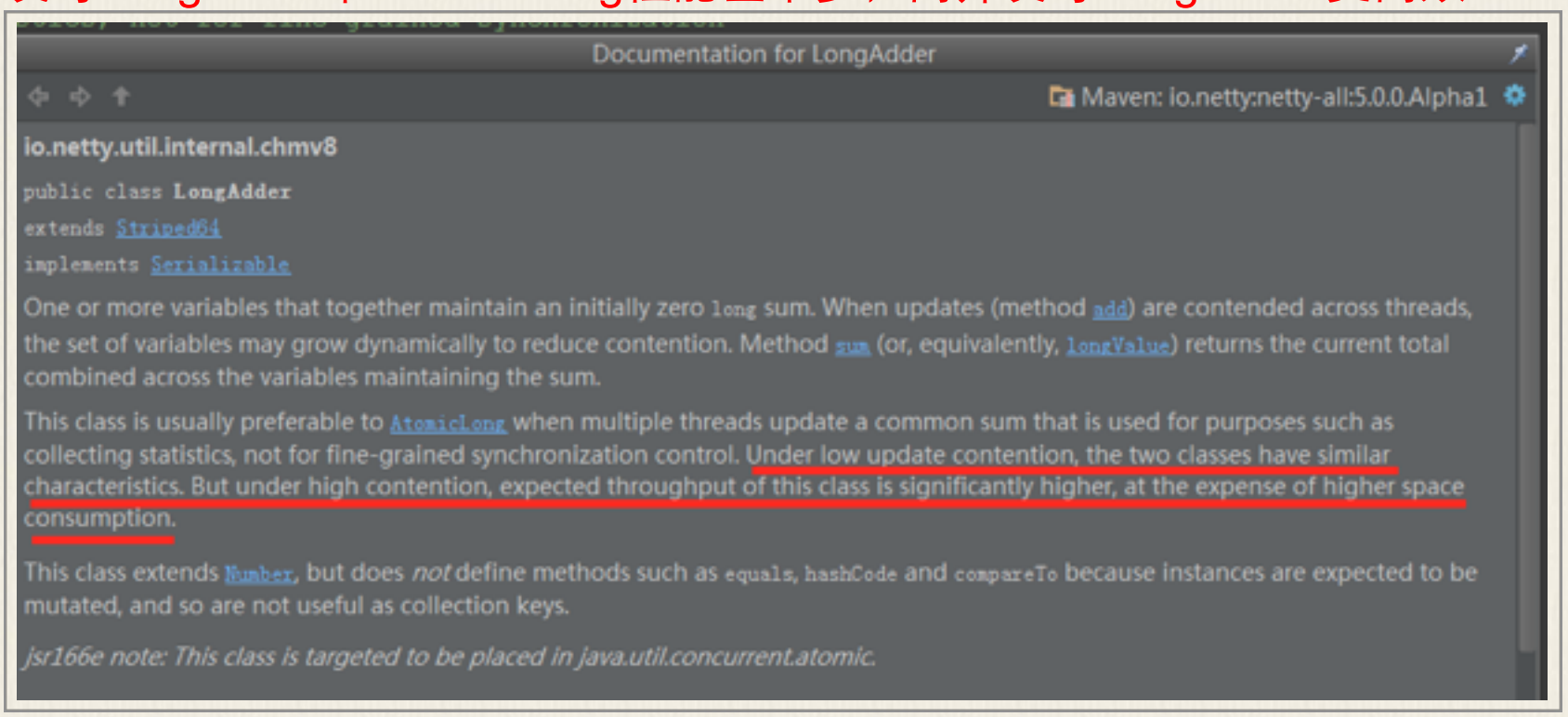
当然，如果as 为null 并且更新失败，才会进入retryUpdate方法。

看到这里我想应该有很多人明白为什么LongAdder会比AtomicLong更高效了，没错，唯一会制约AtomicLong高效的原因是 高并发，高并发意味着CAS的失败几率更高， 重试次数更多，越多线程重试，CAS失败几率又越高，变成恶性循环，AtomicLong效率降低。

那怎么解决？ LongAdder给了我们一个非常容易想到的解决方案: 减少并发，将单一value的更新压力分担到多个value中去，降低单个value的 “热度”，分段更新！！！

这样，线程数再多也会分担到多个value上去更新，只需要增加cell就可以降低 value的 “热度” AtomicLong中的恶性循环不就解决了吗？ cells 就是用来存储这个 “段”的，每个段又叫做cell, cell中的value 就是存放更新值的， 这样，当我需要总数时，把cells 中的value都累加一下不就可以了么！！

当然，聪明之处远远不仅仅这里，在看看add方法中的代码，casBase方法可不可以不要，直接分段更新,上来就计算 索引位置，然后更新value?
答案是不好，不是不行，因为，casBase操作等价于 AtomicLong中的cas操作，要知道，LongAdder中分段更新这样的处理方式是有坏处的，分段操作必然带来空间上的浪费，可以空间换时间， 但是，能不换就不换，要空间时间都节约~！ 所以，casBase操作保证了在低并发时，不会立即进入分支做分段更新操作，因为低并发时，casBase操作基本都会成功，只有并发高到一定程度了， 才会进入分支，所以，Doug Lead对该类的说明是： 低并发时LongAdder和AtomicLong性能差不多，高并发时LongAdder更高效！



但是，Doung Lea 还是没这么简单，聪明之处还没有结束...

虽然如此，retryUpdate中做了什么事，也基本略知一二了，因为cell中的value都更新失败(说明该索引到这个cell的线程也很多，并发也很高时) 或者cells数组为空时才会调用retryUpdate,

因此，retryUpdate里面应该会做两件事:
1. 扩容，将cells数组扩大，降低每个cell的并发量，同样，这也意味着cells数组的rehash动作。
2. 给空的cells变量赋一个新的Cell数组。

是不是这样呢？ 继续看代码:
代码比较长，变成文本看看，为了方便大家看if else 分支，对应的 { } 我在括号后面用分支XXX 标注出来

可以看到，这个时候Doug Lea才愿意使用死循环保证更新成功~!

因为进入这个方法的几率已经非常低了。

```
final void retryUpdate(long x, HashCode hc, boolean wasUncontended) {
    int h = hc.code;
    boolean collide = false;                // True if last slot nonempty
    for (;;) {
        Cell[] as; Cell a; int n; long v;
        if ((as = cells) != null && (n = as.length) > 0) {// 分支1
            if ((a = as[(n - 1) & h]) == null) {
                if (busy == 0) {            // Try to attach new Cell
                    Cell r = new Cell(x);   // Optimistically create
                    if (busy == 0 && casBusy()) {
                        boolean created = false;
                        try {               // Recheck under lock
                            Cell[] rs; int m, j;
                            if ((rs = cells) != null &&
                                    (m = rs.length) > 0 &&
                                    rs[j = (m - 1) & h] == null) {
                                rs[j] = r;
                                created = true;
                            }
                        } finally {
                            busy = 0;
                        }
                          if (created)
                              break;
                        continue;           // Slot is now non-empty
                    }
                }
                collide = false;
            }
            else if (!wasUncontended)       // CAS already known to fail
                wasUncontended = true;      // Continue after rehash
            else if (a.cas(v = a.value, fn(v, x)))
                break;
            else if (n >= NCPU || cells != as)
                collide = false;            // At max size or stale
            else if (!collide)
                collide = true;
            else if (busy == 0 && casBusy()) {
                try {
                    if (cells == as) {      // Expand table unless stale
                        Cell[] rs = new Cell[n << 1];
                        for (int i = 0; i < n; ++i)
                            rs[i] = as[i];
                        cells = rs;
                    }
                } finally {
                    busy = 0;
                }
```

```
                collide = false;
                continue;                    // Retry with expanded table
            }
            h ^= h << 13;                    // Rehash               h ^= h >>> 17;
            h ^= h << 5;
        }
        else if (busy == 0 && cells == as && casBusy()) {//分支2
            boolean init = false;
            try {                           // Initialize table
                if (cells == as) {
                    Cell[] rs = new Cell[2];
                    rs[h & 1] = new Cell(x);
                    cells = rs;
                    init = true;
                }
            } finally {
                busy = 0;
            }
            if (init)
                break;
        }
        else if (casBase(v = base, fn(v, x)))
            break;                          // Fall back on using base
    }
    hc.code = h;                            // Record index for next time
}
```

分支2中，为cells为空的情况，需要new 一个Cell数组。
分支1分支中，略复杂一点点：
注意，几个分支中都提到了busy这个方法，这个可以理解为一个CAS实现的锁，只有在需要更新cells数组的时候才会更新该值为1，如果更新失败，则说明当前有线程在更新cells数组，当前线程需要等待。重试。
回到分支1中，这里首先判断当前cells数组中的索引位置的cell元素是否为空，如果为空，则添加一个cell到数组中。
否则更新 标示冲突的标志位wasUncontended 为 true ，重试。
否则，再次更新cell中的value,如果失败，重试。
。。。。。。。。一系列的判断后，如果还是失败，下下下策，reHash,直接将cells数组扩容一倍，并更新当前线程的hash值，保证下次更新能尽可能成功。
可以看到，LongAdder确实用了很多心思减少并发量，并且，每一步都是在”没有更好的办法“的时候才会选择更大开销的操作，从而尽可能的用最最简单的办法去完成操作。追求简单，但是绝对不粗暴。

————————————分割线————————————-

昨天在coolshell　投稿后 （文章在这里） 和左耳朵耗子简单讨论了下，发现左耳朵耗子对读者思维的引导还是非常不错的，在第一次发现这个类后，对里面的实现又提出了更多的问题，引导大家思考，值得学习，赞一个~

我们 发现的问题有这么几个:

1. jdk 1.7中是不是有这个类?
我确认后，结果如下:　　jdk-7u51版本上 上还没有 但是jdk-8u20 版本上已经有了。代码基本一样 ， 增加了对dou-
ble类型的支持和删除了一些冗余的代码。

2. base有没有参与汇总?
base在调用intValue等方法的时候是会汇总的:

```
public long longValue() {
    return sum();
}

/**
 * Returns the {@link #sum} as an {@code int} after a narr
 * primitive conversion.
 */
public int intValue() {
    return (int)sum();
}

/**
 * Returns the {@link #sum} as a {@code float}
 * after a widening primitive conversion.
 */
public float floatValue() {
    return (float)sum();
}
```

3. base的顺序可不可以调换?
左耳朵耗子,提出了这么一个问题： 在add方法中，如果cells不会为空后，casBase方法一直都没有用了?

因此，我想可不可以调换add方法中的判断顺序，比如，先做casBase的判断，结果是 不调换可能更好，调换后每次都要CAS一下，在高并发时，失败几率非常高，并且是恶性循环，比起一次判断，后者的开销明显小很多，还没有副作用。因此，不调换可能会更好。

4. AtomicLong可不可以废掉?
   我的想法是可以废掉了，因为，虽然LongAdder在空间上占用略大，但是，它的性能已经足以说明一切了，无论是从节约空的角度还是执行效率上，AtomicLong基本没有优势了，具体看这个测试（感谢coolshell读者Lemon的回复）:http://blog.palominolabs.com/2014/02/10/java-8-performance-improvements-longadder-vs-atomiclong/

原文链接：http://www.liuinsect.com/2014/04/15/%E6%AF%94atomiclong%E8%BF%98%E9%AB%98%E6%95%88%E7%9A%84longadder-%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90/

C语言的整型溢出问题

作者：@左耳朵耗子
网站：http://coolshell.cn/
芝兰生于深谷，不以无人而不芳；君子修道立德，不为困穷而改

整型溢出有点老生常谈了，bla, bla, bla… 但似乎没有引起多少人的重视。整型溢出会有可能导致缓冲区溢出，缓冲区溢出会导致各种黑客攻击，比如最近OpenSSL的heartbleed事件，就是 一个buffer overread的事件。在这里写下这篇文章，希望大家都了解一下整型溢出，编译器的行为，以及如何防范，以写出更安全的代码。

什么是整型溢出

C语言的整型问题相信大家并不陌生了。对于整型溢出，分为无符号整型溢出和有符号整型溢出。

对于unsigned整型溢出，C的规范是有定义的——“溢出后的数会以2^(8*sizeof(type))作模运算”，也就是说，如果一个unsigned char（1字符，8bits）溢出了，会把溢出的值与256求模。例如：

```
unsigned char x = 0xff;
printf("%d\n", ++x);
```

上面的代码会输出：0 （因为0xff + 1是256，与2^8求模后就是0）

对于signed整型的溢出，C的规范定义是“undefined behavior”，也就是说，编译器爱怎么实现就怎么实现。对于大多数编译器来说，算得啥就是啥。比如：

```
signed char x =0x7f; //注：0xff就是-1了，因为最高位是1也就是负数了
printf("%d\n", ++x);
```

上面的代码会输出：-128，因为0x7f + 0×01得到0×80，也就是二进制的1000 0000，符号位为1，负数，后面为全0，就是负的最小数，即-128。

另外，千万别以为signed整型溢出就是负数，这个是不定的。比如：

```
signed char x = 0x7f;
```

```
signed char y = 0x05;
signed char r = x * y;
printf("%d\n", r);
```

上面的代码会输出：123

相信对于这些大家不会陌生了。

整型溢出的危害

下面说一下，整型溢出的危害。

示例一：整形溢出导致死循环

```
... ...
... ...
short len = 0;
... ...
while(len< MAX_LEN) {
    len += readFromInput(fd, buf);
    buf += len;
}
```

上面这段代码可能是很多程序员都喜欢写的代码（我在很多代码里看到过多次），其中的MAX_LEN 可能会是个比较大的整型，比如32767，我们知道short是16bits，取值范围是-32768 到 32767 之间。但是，上面的while循环代码有可能会造成整型溢出，而len又是个有符号的整型，所以可能会成负数，导致不断地死循环。

示例二：整形转型时的溢出

```
int copy_something(char *buf, int len)
{
    #define MAX_LEN 256
    char mybuf[MAX_LEN];</pre>
<pre>     ... ...
     ... ...

     if(len > MAX_LEN){ // <---- [1]
         return -1;
     }

     return memcpy(mybuf, buf, len);
}
```

上面这个例子中，还是[1]处的if语句，看上去没有会问题，但是len是个signed int，而memcpy则需一个size_t的len，也就是一个unsigned 类型。

于是，len会被提升为unsigned，此时，如果我们给len传一个负数，会通过了if的检查，但在memcpy里会被提升为一个正数，于是我 们的my-buf就是overflow了。这个会导致mybuf缓冲区后面的数据被重写。

示例三：分配内存

关于整数溢出导致堆溢出的很典型的例子是，OpenSSH Challenge-Response SKEY/BSD_AUTH 远程缓冲区溢出漏洞。下面这段有问题的代码摘自OpenSSH的代码中的auth2-chall.c中的 input_userauth_info_re-sponse() 函数:

```
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp*sizeof(char*));
    for (i = 0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}
```

上面这个代码中，nresp是size_t类型（size_t一般就是unsigned int/long int），这个示例是一个解数据包的示例，一般来说，数据包中都会有一个len，然后后面是data。如果我们精心准备一个len，比 如：1073741825（在32位系统上，指针占4个字节，unsigned int的最大值是0xffffffff，我们只要提供0xffffffff/4 的值——0×40000000，这里我们设置了0×4000000 + 1）， nresp就会读到这个值，然后nresp*sizeof(char*)就成了1073741825 * 4，于是溢出，结果成为了 0×100000004，然后求模，得到4。于是，malloc(4)，于是后面的for循环1073741825 次，就可以干坏事了（经过0×40000001的循环,用户的数据早已覆盖了xmalloc原先分配的4字节的空间以及后面的数据，包括程序代码，函数指 针，于是就可以改写程序逻辑。关于更多的东西，你可以看一下这篇文章《Survey of Protections from Buffer-Overflow Attacks》）。

示例四：缓冲区溢出导致安全问题

```
int func(char *buf1, unsigned int len1,
         char *buf2, unsigned int len2 )
{
   char mybuf[256];

   if((len1 + len2) > 256){    //<--- [1]
       return -1;
   }

   memcpy(mybuf, buf1, len1);
   memcpy(mybuf + len1, buf2, len2);
```

```
    do_some_stuff(mybuf);

    return 0;
}
```

上面这个例子本来是想把buf1和buf2的内容copy到mybuf里，其中怕len1 + len2超过256 还做了判断，但是，如果len1+len2溢出了，根据unsigned的特性，其会与2^32求模，所以，基本上来说，上面代码中的[1]处有可能为假 的。（注：通常来说，在这种情况下，如果你开启-O代码优化选项，那个if语句块就全部被和谐掉了——被编译器给删除了）比如，你可以测试一下 len1=0×104， len2 = 0xfffffffc 的情况。

这样的例子有很多很多，这些整型溢出的问题如果在关键的地方，尤其是在搭配有用户输入的地方，如果被黑客利用了，就会导致很严重的安全问题。

关于编译器的行为

在谈一下如何正确的检查整型溢出之前，我们还要来学习一下编译器的一些东西。请别怪我罗嗦。

编译器优化

如何检查整型溢出或是整型变量是否合法有时候是一件很麻烦的事情，就像上面的第四个例子一样，编译的优化参数-O/-O2/-O3基本上会假设你的程序不会有整形溢出。会把你的代码中检查溢出的代码给优化掉。

关于编译器的优化，在这里再举个例子，假设我们有下面的代码（又是一个相当相当常见的代码）：

```
int len;
char* data;

if (data + len < data){
    printf("invalid len\n");
    exit(-1);
}
```

上面这段代码中，len 和 data 配套使用，我们害怕len的值是非法的，或是len溢出了，于是我们写下了if语句来检查。这段代码在-O的参数下正常。但是在-O2的编译选项下，整个if语句块被优化掉了。

你可以写个小程序，在gcc下编译（我的版本是4.4.7，记得加上-O2和-g参数），然后用gdb调试时，用disass /m命信输出汇编，你会看到下面的结果（你可以看到整个if语句块没有任何的汇编代码——直接被编译器和谐掉了）：

```
        int len = 10;
        char* data = (char *)malloc(len);
   0x00000000004004d4 <+4>:     mov    $0xa,%edi
   0x00000000004004d9 <+9>:     callq  0x4003b8 <malloc@plt>


        if (data + len < data){
            printf("invalid len\n");
            exit(-1);
        }

    }
   0x00000000004004de <+14>:    add    $0x8,%rsp
   0x00000000004004e2 <+18>:    retq
```

对此，你需要把上面 char* 转型成 uintptr_t 或是 size_t，说白了也就是把char*转成unsigned的数据结构，if语句块就无法被优化了。如下所示：

```
if ((uintptr_t)data + len < (uintptr_t)data){
    ... ...
}
```

关于这个事，你可以看一下C99的规范说明《 ISO/IEC 9899:1999 C specification 》第 §6.5.6 页，第8点，我截个图如下：（这段话的意思是定义了指针+/-一个整型的行为，如果越界了，则行为是undefined）

8 When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. In other words, if the expression `P` points to the *i*-th element of an array object, the expressions `(P)+N` (equivalently, `N+(P)`) and `(P)-N` (where `N` has the value *n*) point to, respectively, the *i*+*n*-th and *i*−*n*-th elements of the array object, provided they exist. Moreover, if the expression `P` points to the last element of an array object, the expression `(P)+1` points one past the last element of the array object, and if the expression `Q` points one past the last element of an array object, the expression `(Q)-1` points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary `*` operator that is evaluated.

注意上面标红线的地方，说如果指针指在数组范围内没事，如果越界了就是undefined，也就是说这事交给编译器实现了，编译器想咋干咋干，那怕你想把其优化掉也可以。在这里要重点说一下，C语言中的一个大恶魔——Undefined! 这里都是“野兽出没”的地方，你一定要小心小心再小心。

花絮：编译器的彩蛋

上面说了所谓的undefined行为就全权交给编译器实现，gcc在1.17版本下对于undefined的行为还玩了个彩蛋（参看Wikipedia）。

下面gcc 1.17版本下的遭遇undefined行为时，gcc在unix发行版下玩的彩蛋的源代码。我们可以看到，它会去尝试去执行一些游戏 NetHack， Rogue 或是Emacs的 Towers of Hanoi，如果找不到，就输出一条NB的报错。

```
execl("/usr/games/hack", "#pragma", 0); // try to run the game NetH-
ack
execl("/usr/games/rogue", "#pragma", 0); // try to run the game Rogue
// try to run the Tower's of Hanoi simulation in Emacs.
execl("/usr/new/emacs", "-f","hanoi","9","-kill",0);
execl("/usr/local/emacs","-f","hanoi","9","-kill",0); // same as
above
fatal("You are in a maze of twisty compiler features, all differ-
ent");
```

正确检测整型溢出

在看过编译器的这些行为后，你应该会明白——“在整型溢出之前，一定要做检查，不然，就太晚了”。

我们来看一段代码：

```
void foo(int m, int n)
{
    size_t s = m + n;
    .......
}
```

上面这段代码有两个风险：1）有符号转无符号，2）整型溢出。这两个情况在前面的那些示例中你都应该看到了。所以，你千万不要把任何检查的代码写在 s = m + n 这条语名后面，不然就太晚了。 undefined行为就会出现了——用句纯正的英文表达就是——“Dragon is here”——你什么也控制不住了。（注意：有些初学者也许会以为size_t是无符号的，而根据优先级 m 和 n 会被提升到unsigned int。其实不是这样的，m 和 n 还是signed int，m + n 的结果也是signed int，然后再把这个结果转成unsigned int 赋值给s）

比如，下面的代码是错的：

```
void foo(int m, int n)
{
    size_t s = m + n;
    if ( m>0 && n>0 && (SIZE_MAX - m < n) ){
        //error handling...
    }
```

```
}
```

上面的代码中，大家要注意 (SIZE_MAX – m < n) 这个判断，为什么不用m + n > SIZE_MAX呢？因为，如果 m + n 溢出后，就被截断了，所以表达式恒真，也就检测不出来了。另外，这个表达式中，m和n分别会被提升为unsigned。

但是上面的代码是错的，因为：

1）检查的太晚了，if之前编译器的undefined行为就已经出来了（你不知道什么会发生）。

2）就像前面说的一样，(SIZE_MAX – m < n) 可能会被编译器优化掉。

3）另外，SIZE_MAX是size_t的最大值，size_t在64位系统下是64位的，严谨点应该用INT_MAX或是UINT_MAX

所以，正确的代码应该是下面这样：

```
void foo(int m, int n)
{
    size_t s = 0;
    if ( m>0 && n>0 && ( UINT_MAX - m < n ) ){
        //error handling...
        return;
    }
    s = (size_t)m + (size_t)n;
}
```

在《苹果安全编码规范》（PDF）中，第28页的代码中：

如果n和m都是signed int，那么这段代码是错的。正确的应该像上面的那个例子一样，至少要在n*m时要把 n 和 m 给 cast 成 size_t。因为，n*m可能已经溢出了，已经undefined了，undefined的代码转成size_t已经没什么意义了。（如果m和n是 unsigned int，也会溢出），上面的代码仅在m和n是size_t的时候才有效。

不管怎么说，《苹果安全编码规范》绝对值得你去读一读。

上溢出和下溢出的检查

前面的代码只判断了正数的上溢出overflow，没有判断负数的下溢出underflow。让们来看看怎么判断：

对于加法，还好。

```
#include <limits.h>
```

```
void f(signed int si_a, signed int si_b) {
    signed int sum;
    if (((si_b > 0) && (si_a > (INT_MAX - si_b))) ||
        ((si_b < 0) && (si_a < (INT_MIN - si_b)))) {
        /* Handle error */
        return;
    }
    sum = si_a + si_b;
}
```

对于乘法，就会很复杂（下面的代码太夸张了）：

```
void func(signed int si_a, signed int si_b)
{
  signed int result;
  if (si_a > 0) {  /* si_a is positive */
    if (si_b > 0) {  /* si_a and si_b are positive */
      if (si_a > (INT_MAX / si_b)) {
        /* Handle error */
      }
    } else { /* si_a positive, si_b nonpositive */
      if (si_b < (INT_MIN / si_a)) {
        /* Handle error */
      }
    } /* si_a positive, si_b nonpositive */
  } else { /* si_a is nonpositive */
    if (si_b > 0) { /* si_a is nonpositive, si_b is positive */
      if (si_a < (INT_MIN / si_b)) {
        /* Handle error */
      }
    } else { /* si_a and si_b are nonpositive */
      if ( (si_a != 0) && (si_b < (INT_MAX / si_a))) {
        /* Handle error */
      }
    } /* End if si_a and si_b are nonpositive */
  } /* End if si_a is nonpositive */

  result = si_a * si_b;
}
```

更多的防止在操作中整型溢出的安全代码可以参看《INT32-C. Ensure that operations on signed integers do not result in overflow》

其它

对于C++来说，你应该使用STL中的numeric_limits::max() 来检查溢出。

另外，微软的SafeInt类是一个可以帮你远理上面这些很tricky的类，下载地址：http://safeint.codeplex.com/

对于Java 来说，一种是用JDK 1.7中Math库下的safe打头的函数，如safeAdd()和safeMultiply()，另一种用更大尺寸的数据类型，最大可以到BigInteger。

可见，写一个安全的代码并不容易，尤其对于C/C++来说。对于黑客来说，他们只需要搜一下开源软件中代码有memcpy/strcpy之类的地方，然后看一看其周边的代码，是否可以通过用户的输入来影响，如果有的话，你就惨了。

参考：

- Basic Integer Overflow
- OWASP：Integer overflow
- C compilers may silently discard some wraparound checks
- Apple Secure Coding Guide
- Wikipedia: Undefined Behavior
- INT32-C. Ensure that operations on signed integers do not result in overflow

最后， 不好意思，这篇文章可能罗嗦了一些，大家见谅。

原文链接：http://coolshell.cn/articles/11466.html

我为什么放弃Go语言

作者：庄晓立（Liigo）
网站：http://my.csdn.net/liigo
当我沉默着的时候，我觉得充实；我将开口，同时感到空虚……

有好几次，当我想起来的时候，总是会问自己：我为什么要放弃Go语言？这个决定是正确的吗？是明智和理性的吗？其实我一直在认真思考这个问题。

开门见山地说，我当初放弃Go语言（golang），就是因为两个“不爽”：第一，对Go语言本身不爽；第二，对Go语言社区里的某些人不爽。毫无疑问，这是非常主观的结论。但是我有足够详实的客观的论据，支撑这个看似主观的结论。

第0节：我的Go语言经历

先说说我的经历吧，以避免被无缘无故地当作Go语言的低级黑。

2009年底，Go语言（golang）第一个公开版本发布，笼罩着“Google公司制造”的光环，吸引了许多慕名而来的尝鲜者，我 （Liigo）也身居其中，笼统的看了一些Go语言的资料，学习了基础的教程，因对其语法中的分号和花括号不满，很快就遗忘掉了，没拿它当一回事。

两年之后，2011年底，Go语言发布1.0的计划被提上日程，相关的报道又多起来，我再次关注它，[重新评估][1]之后决定深入参与Go语言。 我订阅了其users、nuts、dev、commits等官方邮件组，坚持每天阅读其中的电子邮件，以及开发者提交的每一次源代码更新，给Go提交了许 多改进意见，甚至包括[修改Go语言编译器源代码][2]直接参与开发任务。如此持续了数月时间。

到2012年初，Go 1.0发布，语言和标准库都已经基本定型，不可能再有大幅改进，我对Go语言未能在1.0定型之前更上一个台阶、实现自我突破，甚至带着诸多明显缺陷走向 1.0，感到非常失望，因而逐渐疏远了它（所以Go 1.0之后的事情我很少关心）。后来看到即将发布的Go 1.1的Release

Note，发现语言层面没有太大改变，只是在库和工具层面有所修补和改进，感到它尚在幼年就失去成长的动力，越发失望。外加Go语言社区里的某些人，其中 也包括Google公司负责开发Go语言的某些人，其态度、言行，让我极度厌恶，促使我决绝地离弃Go语言。

[1]: https://plus.google.com/+LiigoZhuang/posts/CpRNPeDXUDW

[2]: http://blog.csdn.net/liigo/article/details/7467309

第1节：我为什么对Go语言不爽?

Go语言有很多让我不爽之处，这里列出我现在还能记起的其中一部分，排名基本上不分先后。读者们耐心地看完之后，还能淡定地说一句“我不在乎”吗?

1.1 不允许左花括号另起一行

关于对花括号的摆放，在C语言、C++、Java、C#等社区中，十余年来存在持续争议，从未形成一致意见。在我看来，这本来就是主观倾向很重的抉择，不违反原则不涉及是非的情况下，不应该搞一刀切，让程序员或团队自己选择就足够了。编程语言本身强行限制，把自己的喜好强加给别人，得不偿失。无论倾 向于其中任意一种，必然得罪与其对立的一群人。虽然我现在已经习惯了把左花括号放在行尾，但一想到被禁止其他选择，就感到十分不爽。Go语言这这个问题 上，没有做到“团结一切可以团结的力量”不说，还有意给自己树敌，太失败了。

1.2 编译器莫名其妙地给行尾加上分号

对Go语言本身而言，行尾的分号是可以省略的。但是在其编译器（gc）的实现中，为了方便编译器开发者，却在词法分析阶段强行添加了行尾的分号，反 过来又影响到语言规范，对“怎样添加分号”做出特殊规定。这种变态做法前无古人。在左花括号被意外放到下一行行首的情况下，它自动在上一行行尾添加的分 号，会导致莫名其妙的编译错误（Go 1.0之前），连它自己都解释不明白。如果实在处理不好分号，干脆不要省略分号得了；或者，Scala和JavaScript的编译器是开源的，跟它们学 学怎么处理省略行尾分号可以吗?

1.3 极度强调编译速度，不惜放弃本应提供的功能

程序员是人不是神，编码过程中免不了因为大意或疏忽犯一些错。其中有一些，是大家集体性的很容易就中招的错误（Go语言里的例子我暂时想不起

来，C++里的例子有“基类析构函数不是虚函数”）。这时候编译器应该站出来，多做一些检查、约束、核对性工作，尽量阻止常规错误的发生，尽量不让有潜在 错误的代码编译通过，必要时给出一些警告或提示，让程序员留意。编译器不就是机器么，不就是应该多做脏活累活杂活、减少人的心智负担么？编译器多做一项检 查，可能会避免数十万程序员今后多年内无数次犯同样的错误，节省的时间不计其数，这是功德无量的好事。但是Go编译器的作者们可不这么想，他们不愿意自己 多花几个小时给编译器增加新功能，觉得那是亏本，反而减慢了编译速度。他们以影响编译速度为由，拒绝了很多对编译器改进的要求。典型的因噎废食。强调编译 速度固然值得赞赏，但如果因此放弃应有的功能，我不赞成。

1.4 错误处理机制太原始

在Go语言中处理错误的基本模式是：函数通常返回多个值，其中最后一个值是error类型，用于表示错误类型极其描述；调用者每次调用完一个函数， 都需要检查这个error并进行相应的错误处理。这种模式跟C语言那种很原始的错误处理相比如出一辙，并无实质性改进。实际应用中很容易形成多层嵌套的 if else语句，可以想一想这个编码场景：先判断文件是否存在，如果存在则打开文件，如果打开成功则读取文件，如果读取成功再写入一段数据，最后关闭文件， 别忘了还要处理每一步骤中出现错误的情况，这代码写出来得有多变态、多丑陋？实践中普遍的做法是，判断操作出错后提前return，以避免多层花括号嵌 套，但这么做的后果是，许多错误处理代码被放在前面突出的位置，常规的处理逻辑反而被掩埋到后面去了。而且，error对象的标准接口只能返回一个错误文 本，有时候调用者为了区分不同的错误类型，甚至需要解析该文本。除此之外，你只能手工强制转换error类型到特定子类型。至于panic - recover机制，致命的缺陷是不能跨越库的边界使用，注定是一个半成品，最多只能在自己的pkg里面玩一玩。Java的异常处理虽然也有自身的问题 （比如Checked Exceptions），但总体上还是比Go的错误处理高明很多。

1.5 垃圾回收器（GC）不完善、有重大缺陷

在Go 1.0前夕，其垃圾回收器在32位环境下有内存泄漏，一直拖着不肯改进，这且不说。Go语言垃圾回收器真正致命的缺陷是，会导致整个进程不可预知的间歇性 停顿。像某些大型后台服务程序，如游戏服务器、APP容器等，由于占用内存巨大，其内存对象数量极多，GC完成一次回收周期，可能需要数秒甚至更长时间， 这段时间内，整个服务进程是阻塞的、停顿的，在外界看来就是服务中断、无响应，再牛逼的并发机制到了这里统统失

效。垃圾回收器定期启动，每次启动就导致短 暂的服务中断，这样下去，还有人敢用吗？这可是后台服务器进程，是Go语言的重点应用领域。以上现象可不是我假设出来的，而是事实存在的现实问题，受其严 重困扰的也不是一家两家了（截止到2014年初）。在实践中，你必须努力减少进程中的对象数量，以便把GC导致的间歇性停顿控制在可接受范围内。除此之外你别无选择（难道你还想自己更换GC算法、甚至砍掉GC？那还是Go语言吗？）。跳出圈外，我近期一直在思考，一定需要垃圾回收器吗？没有垃圾回收器就一 定是历史的倒退吗？（可能会新写一篇博客文章专题探讨。）

1.6 禁止未使用变量和多余import

Go编译器不允许存在被未被使用的变量和多余的import，如果存在，必然导致编译错误。但是现实情况是，在代码编写、重构、调试过程中，例如，临时性的注释掉一行代码，很容易就会导致同时出现未使用的变量和多余的import，直接编译错误了，你必须相应的把变量定义注释掉，再翻页回到文件首部 把多余的import也注释掉，……等事情办完了，想把刚才注释的代码找回来，又要好几个麻烦的步骤。还有一个让人蛋疼的问题，编写数据库相关的代码时， 如果你import某数据库驱动的pkg，它编译给你报错，说不需要import这个未被使用的pkg；但如果你听信编译器的话删掉该import，编译 是通过了，运行时必然报错，说找不到数据库驱动；你看看程序员被折腾的两边不是人，最后不得不请出大神_。对待这种问题，一个比较好的解决方案是，视其为编译警告而非编译错误。但是Go语言开发者很固执，不容许这种折中方案。

1.7 创建对象的方式太多令人纠结

创建对象的方式，调用new函数、调用make函数、调用New方法、使用花括号语法直接初始化结构体，你选哪一种？不好选择，因为没有一个固定的模式。从实践中看，如果要创建一个语言内置类型（如channel、map）的对象，通常用make函数创建；如果要创建标准库或第三方库定义的类型的对 象，首先要去文档里找一下有没有New方法，如果有就最好调用New方法创建对象，如果没有New方法，则退而求其次，用初始化结构体的方式创建其对象。 这个过程颇为周折，不像C++、Java、C#那样直接new就行了。

1.8 对象没有构造函数和析构函数

没有构造函数还好说，毕竟还有自定义的New方法，大致也算是构造函数了。没有析构函数就比较难受了，没法实现RAII。额外的人工处理资源清理

工 作，无疑加重了程序员的心智负担。没人性啊，还嫌我们程序员加班还少吗？C++里有析构函数，Java里虽然没有析构函数但是有人家finally语句啊，Go呢，什么都没有。没错，你有个defer，可是那个defer问题更大，详见下文吧。

1.9 defer语句的语义设定不甚合理

Go语言设计defer语句的出发点是好的，把释放资源的“代码”放在靠近创建资源的地方，但把释放资源的“动作”推迟（defer）到函数返回前 执行。遗憾的是其执行时机的设置似乎有些不甚合理。设想有一个需要长期运行的函数，其中有无限循环语句，在循环体内不断的创建资源（或分配内存），并用 defer语句确保释放。由于函数一直运行没有返回，所有defer语句都得不到执行，循环过程中创建的大量短暂性资源一直积累着，得不到回收。而且，系 统为了存储defer列表还要额外占用资源，也是持续增加的。这样下去，过不了多久，整个系统就要因为资源耗尽而崩溃。像这类长期运行的函数，http.ListenAndServe()就是典型的例子。在Go语言重点应用领域，可以说几乎每一个后台服务程序都必然有这么一类函数，往往还都 是程序的核心部分。如果程序员不小心在这些函数中使用了defer语句，可以说后患无穷。如果语言设计者把defer的语义设定为在所属代码块结束时（而 非函数返回时）执行，是不是更好一点呢？可是Go 1.0早已发布定型，为了保持向后兼容性，已经不可能改变了。小心使用defer语句！一不小心就中招。

1.10 许多语言内置设施不支持用户定义的类型

for in、make、range、channel、map等都仅支持语言内置类型，不支持用户定义的类型(?)。用户定义的类型没法支持for in循环，用户不能编写像make、range那样“参数类型和个数”甚至“返回值类型和个数”都可变的函数，不能编写像channel、map那样类似 泛型的数据类型。语言内置的那些东西，处处充斥着斧凿的痕迹。这体现了语言设计的局限性、封闭性、不完善性，像是新手作品——且不论其设计者和实现者如何 权威。

1.11 没有泛型支持，常见数据类型接口丑陋

没有泛型的话，List、Set、Tree这些常见的基础性数据类型的接口就只能很丑陋：放进去的对象是一个具体的类型，取出来之后成了无类型的interface{}（可 以视为所有类型的基础类型），还得强制类型转换之后才能继续使用，令人无语。Go语言缺少min、max这类函数，求数值绝对值的函数abs只接收/返回 双精度小数类型，排序接口只能借助sort.Interface无奈的回避了被比较对象的类型，等等等等，都是没有泛型导致的结果。没有泛

型，接口很难优 雅起来。Go开发者没有明确拒绝泛型，只是说还没有找到很好的方法实现泛型（能不能学学已经开源的语言呀）。现实是，Go 1.0已经定型，泛型还没有，那些丑陋的接口为了保持向后兼容必须长期存在着。

1.12 实现接口不需要明确声明

这一条通常是被当作Go语言的优点来宣传的。但是也有人不赞同，比如我。如果一个类型用Go语言的方式默默的实现了某个接口，使用者和代码维护者都 很难发现这一点（除非仔细核对该类型的每一个方法的函数签名，并跟所有可能的接口定义相互对照），自然也想不到与该接口有关的应用，显得十分隐晦，不直 观。支持者可能会辩解说，我可以在文档中注明它实现了哪些接口。问题是，写在文档中，还不如直接写到类型定义上呢，至少还能得到编译器的静态类型检查。缺 少了编译器的支持，当接口类型的函数签名被改变时，当实现该接口的类型方法被无意中改变时，实现者可能很难意识到，该类型实现该接口的隐含约束事实上已经 被打破了。又有人辩解说，我可以通过单元测试确保类型正确实现了接口呀。我想说的是，明明可以通过明确声明实现接口，享受编译器提供的类型检查，你却要自己找麻烦，去写原本多余的单元测试，找虐很爽吗？Go语言的这种做法，除了减少一些对接口所在库的依赖之外，没有其他好处，得不偿失。

1.13 省掉小括号却省不掉花括号

Go语言里面的if语句，其条件表达式不需要用小括号扩起来，这被作为“代码比较简洁”的证据来宣传。可是，你省掉了小括号，却不能省掉大括号啊，一条完整的if语句至少还得三行吧，人家C、C++、Java都可以在一行之内搞定的（可以省掉花括号）。人家还有x?a:b表达式呢，也是一行搞定，你Go语言用if else写至少得五行吧？哪里简洁了？

1.14 编译生成的可执行文件尺寸非常大

记得当年我写了一个很简单的程序，把所有系统环境变量的名称和值输出到控制台，核心代码也就那么三五行，结果编译出来把我吓坏了：EXE文件的大小 超过4MB。如果是C语言写的同样功能的程序，0.04MB都是多的。我把这个信息反馈到官方社区，结果人家不在乎。是，我知道现在的硬盘容量都数百 GB、上TB了……可您这种优化程度……怎么让我相信您在其他地方也能做到不错呢。（再次强调一遍，我所有的经验和数据都来自Go 1.0发布前夕。）

1.15 不支持动态加载类库

静态编译的程序当然是很好的，没有额外的运行时依赖，部署时很方便。但是之前我们说了，静态编译的文件尺寸很大。如果一个软件系统由多个可执行程序 构成，累加起来就很可观。如果用动态编译，发布时带同一套动态库，可以节省很多容量。更关键的是，动态库可以运行时加载和卸载，这是静态库做不到的。还有 那些LGPL等协议的第三方C库受版权限制是不允许静态编译的。至于动态库的版本管理难题，可以通过给动态库内的所有符号添加版本号解决。无论如何，应该 给予程序员选择权，让他们自己决定使用静态库还是动态库。一刀切的拒绝动态编译是不合适的。

1.16 其他

- 不支持方法和函数重载（overload）
- 导入pkg的import语句后边部分竟然是文本（import ”fmt”）
- 没有enum类型，全局性常量难以分类，iota把简单的事情复杂化
- 定义对象方法时，receiver类型应该选用指针还是非指针让人纠结
- 定义结构体和接口的语法稍繁，interface XXX{} struct YYY{} 不是更简洁吗？前面加上type关键字显得罗嗦。
- 测试类库testing里面没有AssertEqual函数，标准库的单元测试代码中充斥着if a != b { t.Fatal(...) }。
- 语言太简单，以至于不得不放弃很多有用的特性，“保持语言简单”往往成为拒绝改进的理由。
- 标准库的实现总体来说不甚理想，其代码质量大概处于“基本可用”的程度，真正到企业级应用领域，往往就会暴露出诸多不足之处。
- 版本都发展到1.2了，goroutine调度器依旧默认仅使用一个系统线程。GOMAXPROCS的长期存在似乎暗示着官方从来没有足够的信心，让调度器安全正确的运行在多核环境中。这跟Go语言自身以并发为核心的定位有致命的矛盾。

上面列出的是我目前还能想到的对Go语言的不爽之处，毕竟时间过去两年多，还有一些早就遗忘了。其中一部分固然是小不爽，可能忍一忍就过去了，但是很多不爽积累起来，总会时不时地让人难受，时间久了有自虐的感觉。程序员的工作生活本来就够枯燥的，何必呢。

必须要说的是，对于其中大多数不爽之处，我（Liigo）都曾经试图改变过它们：在Go 1.0版本发布之前，我在其官方邮件组提过很多意见和建议，极力据理力争，可以说付出很大努力，目的就是希望定型后的Go语言是一个相对完善的、没有明显 缺陷的编程语言。结果是令人失望的，我人微言轻、势单力薄，不可能影响整个语言的发展走向。1.0之前，最佳的否定自

我、超越自我的机会，就这么遗憾地错 过了。我最终发现，很多时候不是技术问题，而是技术人员的问题。

第2节：我为什么对Go语言的某些人不爽?

这里提到的“某些人”主要是两类：一、负责专职开发Go语言的Google公司员工；二、Go语言的推崇者和脑残粉丝。我跟这两类人打过很多交道，不胜其烦。再次强调一遍，我指的是“某些”人，而不是所有人，请不要对号入座。

Google公司内部负责专职开发Go语言的核心开发组某些成员，他们倾向于闭门造车，固执己见，对第三方提出的建议不重视。他们常常挂在嘴边的口头禅是：现有的做法很好、不需要那个功能、我们开发Go语言是给Google自己用的、Google不需要那个功能、如果你一定要改请fork之后自己 改、别干提意见请提交代码。很多言行都是“反开源”的。通过一些具体的例子，还能更形象的看清这一层。就留下作为课后作业吧。

我最不能接受的就是他们对1.0版本的散漫处理。那时候Go还没到1.0，初出茅庐的小学生，有很大的改进空间，是全面翻新的最佳时机，彼时不改更待何时？1.0是打地基的版本，基础不牢靠，等1.0定型之后，处处受到向后兼容性的牵制，束手缚脚，每前进一步都阻力重重。急于发布1.0，过早定型， 留下诸多遗憾，彰显了开发者的功利性强，在技术上不追求尽善尽美。

Go语言的核心开发成员，他们日常的开发工作是使用C语言——Go语言的编译器和运行时库，包括语言核心数据结构map、channel、 scheduler，都是C开发的——真正用自己开发的Go语言进行实际的大型应用开发的机会并不多，虽然标准库是用Go语言自己写的，但他们却没有大范 围使用标准库的经历。实际上，他们缺少使用Go语言的实战开发经验，往往不知道处于开发第一线的用户真正需要什么，无法做到设身处地为程序员着想。缺少使 用Go语言的亲身经历，也意味着他们不能在日常开发中，及时发现和改进Go语言的不足。这也是他们往往自我感觉良好的原因。

Go语言社区里，有一大批Go语言的推崇者和脑残粉丝，他们满足于现状，不思进取，处处维护心中的“神”，容不得批评意见，不支持对语言的改进要求。当年我对Go语言的很多批评和改进意见，极少得到他们的支持，他们不但不支持还给予打击，我就纳闷了，他们难道不希望Go语言更完善、更优秀吗？我后 来才意识到，他们跟乔帮主的苹果脑残粉丝们，言行一脉相承，具有极端宗教倾向，神化主子、打击异己真是不遗余力呀。简简单单的

技术问题，就能被他们上升到 意识形态之争。现实的例子是蛮多的，有兴趣的到网上去找吧。正是因为他们的存在，导致更多理智、清醒的Go语言用户无法真正融入整个社区。

如果一个项目、团队、社区，到处充斥着赞美、孤芳自赏、自我满足、不思进取，排斥不同意见，拒绝接纳新方案，我想不到它还有什么前进的动力。逆水行舟，是不进反退的。

第3节：还有比Go语言更好的选择吗?

我始终坚持一个颇有辩证法意味的哲学观点：在更好的替代品出现之前，现有的就是最好的。失望是没有用的，抱怨是没有用的，要么接受，要么逃离。我曾 经努力尝试过接受Go语言，失败之后，注定要逃离。发现更好的替代品之后，无疑加速了逃离过程。还有比Go语言更好的替代品吗？当然有。作为一个屌丝程序 员，我应该告诉你它是什么，但是我不说。现在还不是时候。我现在不想把这两门编程语言对立起来，引发另一场潜在的语言战争。这不是此文的本意。如果你非要 从现有信息中推测它是什么，那完全是你自己的事。如果你原意等，它或许很快会浮出水面，也未可知。

第4节：写在最后

我不原意被别人代表，也不愿意代表别人。这篇文章写的是我，一个叫Liigo的80后屌丝程序员，自己的观点。你完全可以主观地认为它是主观的，也完全可以客观地以为它是客观的，无论如何，那是你的观点。

这篇文字是从记忆里收拾出来的。有些细节虽可考，而不值得考。——我早已逃离，不愿再回到当年的场景。文中涉及的某些细节，可能会因为些许偏差，影响其准确性；也可能会因为缺少出处，影响其客观性。如果有人较真，非要去核实，我相信那些东西应该还在那里。

Go语言也非上文所述一无是处，它当然有它的优势和特色。读者们判断一件事物，应该是优劣并陈，做综合分析，不能单听我一家负面之言。但是它的那些不爽之处，始终让我不爽，且不能从其优秀处得以完全中和，这是我不得不放弃它的原因。

原文链接：http://blog.csdn.net/liigo/article/details/23699459

微博关系服务与Redis的故事

作者：@唐福林
微博技术委员会成员，微博平台资深架构师，致力于高性能高可用互联网服务开发，及高效率团队建设。

新浪微博的工程师们曾经在多个公开场合都讲到过，微博平台当前在使用并维护着可能是世界上最大的Redis集群，其中最大的一个业务，单个业务使用了超过 10T 的内存，这里说的就是微博关系服务。

风起

2009年微博刚刚上线的时候，微博关系服务使用的是最传统的 Memcache+Mysql 的方案。Mysql 按 uid hash 进行了分库分表，表结构非常简单：

tid	fromuid	touid	addTime
自增id	关系主体	关系客体	加关注时间

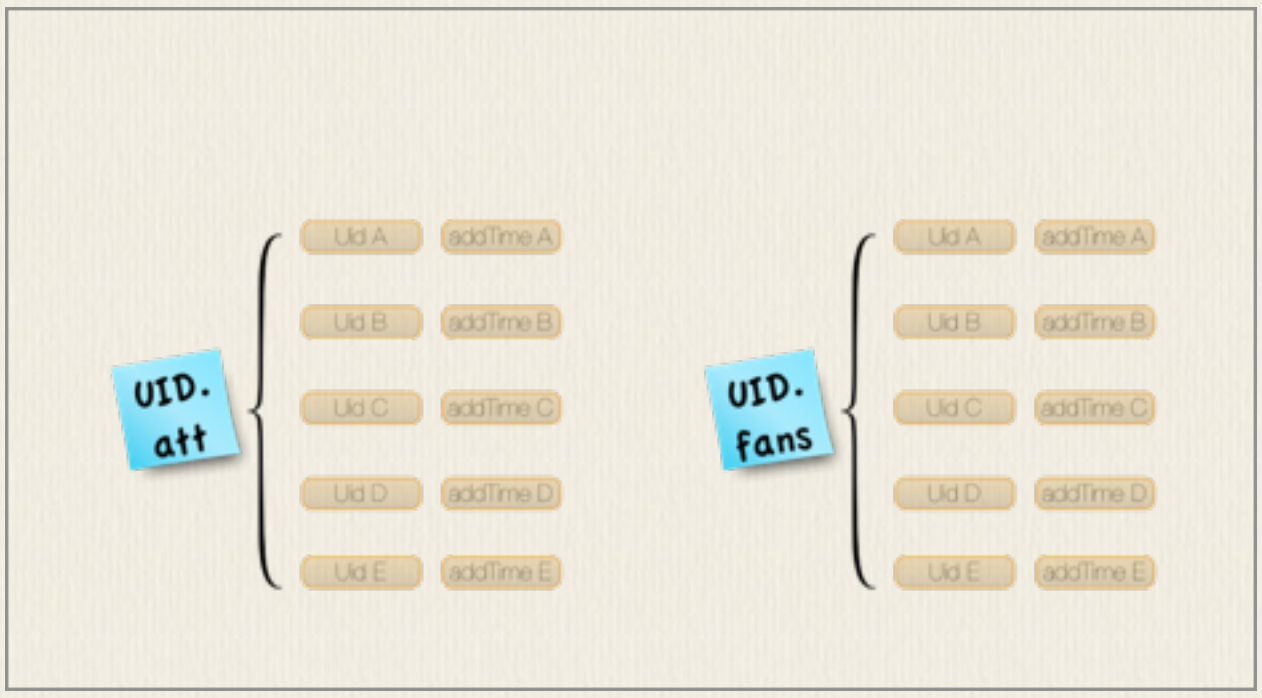
业务方存在两种查询：

- 查询用户的关注列表：select touid from table where fromuid=? order by addTime desc
- 查询用户的粉丝列表：select fromuid from table where touid=? order by addTime desc

两种查询的业务需求与分库分表的架构设计存在矛盾，最终导致了冗余存储：以 fromuid 为hash key存一份，以 touid 为hash key再存一份。memcache key 为 fromuid.suffix，使用不同的 suffix 来区分是关注列表还是粉丝列表，cache value 则为 PHP Serialize 后的 Array。后来为了优化性能，将 value 换成了自己拼装的 byte 数组。

云涌

2011年微博进行平台化改造过程中，业务提出了新的需求：在核心接口中增加了“判断两个用户的关系”的步骤，并增加了“双向关注”的概念。因此两个用户的关系存在四种状态：关注，粉丝，双向关注和无任何关系。为了高效的实现这个需求，平台引入了 Redis 来存储关系。平台使用 Redis 的 hash 来存储关系：key 依然是 uid.suffix，关注列表，粉丝列表及双向关注列表各自有一个不同的 suffix，value 是一个hash，field 是 touid，value 是 addTime。order by addTime 的功能则由 Service 内部 sort 实现。部分大V的粉丝列表可能很长，与产品人员的沟通协商后，将存储限定为“最新的5000个粉丝列表”。



微博关系存储Redis结构

需求实现：

- 查询用户关注列表：hgetAll uid.following ，then sort
- 查询用户粉丝列表：hgetAll uid.follower，then sort
- 查询用户双向关注列表：hgetAll uid.bifollow，then sort
- 判断两个用户关系：hget uidA.following uidB && hget uidB.following uidA

后来又增加了几个更复杂的需求：“我与他的共同关注列表”、“我关注的人里谁关注了他”等等，就不展开来讲了。

平台在刚引入 Redis 的一段时间里踩了不少坑，举几个例子：

- 运维工具和流程从零开始做，运维成熟的速度赶不上业务增长的速度：在还没来得及安排性能调优的工作，fd 已经达到默认配置的上限了，最后我们只能趁凌晨业务低峰期重启 Redis 集群，以便设置新的 ulimit 参数
- 平台最开始使用的 Redis 版本是 2.0，因为 Redis 代码足够简单，从引入到微博起，我们就开始对其进行了定制化开发，从主从复制，到写磁盘限速，再到内存管理，都进行了定制。导致的结果是，有一段时间，微 博的线上存在超过5种不同的 Redis 修改版，对于运维，bug-fix，升级都带来了巨大的麻烦。后来由田风军 @果爸果爸 为内部 Redis 版本提供了不停机升级功能后，才慢慢好转。
- 平台有一个业务曾经使用了非默认 db ，后来费了好大力气去做迁移
- 平台还有一个业务需要定期对数据进行 flush db ，以腾出空间存储最新数据。为了避免在 flush db 阶段影响线上业务，我们从 client 到 server 都做了大量的修改。
- 平台每年长假前都会做一些线上业务排查，和故障模拟（2013年甚至做了一个名叫 Touchstone 的容灾压测系统）。2011年十一假前，我们用 iptables 将 Redis 端口的所有包都 drop 掉，结果 client 端等了 120 秒才返回。于是我们在放假前熬夜加班给 client 添加超时检测功能，但真正上线还是等到了假期回来后。

破茧

对于微博关系服务，最大的挑战还是容量和访问量的快速增长，这给我们的 Redis 方案带来了不少的麻烦：

第一个碰到的麻烦是 Redis 的 hgetAll 在 hash size 较大的场景下慢请求比例较高。我们调整了 hash-max-zip-size，节约了1/3的内存，但对业务整体性能的提升有限。最后，我们不得不在 Redis 前面又挡了一层 memcache，用来抗 hgetAll 读的问题。

第二个麻烦是新上的需求：“我关注的人里谁关注了他”，由于用户的粉丝列表可能不全，在这种情况下就不能用关注列表与粉丝列表求交集的方式来计算结 果，只能降级到需求的字面描述步骤：取我的关注人列表，然后逐个判断这些人里谁关注了他。client 端分批并行发起请求，还好 Redis 的单个关系判断非常快。

第三个麻烦，也是最大的麻烦，就是容量增长的问题了。最初的设计方案，按 uid hash 成 16 个端口，每台 64G 内存的机器上部署 2 个端口，每个业务 IDC 机房部署一套。后来，每台机器上就只部署一个端口了。再后来，128G 内存的机器还没有进入公司采购目录，64G 内存就即将 OOM 了，所以我们不得不做了一次端口扩容：16端口拆64端口，依然是每台 64G 内存机器上部署 2 个端口。再后来，又只部署一个端口。再后来，升级到 128G 内存机器。再后来，128G 机器上出现 OOM 了！现在怎么办?

化蝶

为了从根本上解决容量的问题，我们开始寻找一种本质的解决方案。最初选择引入 Redis 作为一个 storage，是因为用户关系判断功能请求的数据热点不是很集中，长尾效果明显，cache miss 可能会影响核心接口性能，而保证一个可接受的 cache 命中率，耗费的内存与 storage 差别不大。但微博经过了 3 年的演化，最初作为选择依据的那些假设前提，数据指标都已经发生了变化：随着用户基数的增大，冷用户的绝对数量也在增大；Redis 作为存储，为了数据可靠性必须开启 rdb 和 aof，而这会导致业务只能使用一半的机器内存；Redis hash 存储效率太低，特别是与内部极度优化过的 RedisCounter 对比。种种因素加在一起，最终确定下来的方向就是：将 Redis 在这里的 storage 角色降低为 cache 角色。

前面提到的微博关系服务当前的业务场景，可以归纳为两类：一类是取列表，一类是判断元素在集合中是否存在，而且是批量的。即使是 Redis 作为 storage 的时代，取列表都要依赖前面的 memcache 帮忙抗，那么作为 cache 方案，取列表就全部由 memcache 代劳了。批量判断元素在集合中是否存在，redis hash 依然是最佳的数据结构，但存在两个问题：cache miss 的时候，从 db 中获取数据后，set cache 性能太差：对于那些关注了 3000 人的微博会员们，set cache 偶尔耗时可达到 10ms 左右，这对于单线程的 Redis 来说是致命的，意味着这 10ms 内，这个端口无法提供任何其它的服务。另一个问题是 Redis hash 的内存使用效率太低，对于目标的 cache 命中率来说，需要的 cache 容量还是太大。于是，我们又祭出 “Redis定制化”的法宝：将 redis hash 替换成一个“固定长度开放hash寻址数组”，在 Redis 看来就是一个 byte 数组，set cache 只需要一次 redis set。通过精心选择的 hash 算法及数组填充率，能做到批量判断元素是否存在的性能与原生的 redis hash 相当。

通过微博关系服务 Redis storage 的 cache 化改造，我们将这里的 Redis 内存占用降低了一个数量级。它可能会失去“最大的单个业务Redis集群”的头衔，但我们比以前更有成就感，更快乐了。

作者简介

唐福林（@唐福林），微博技术委员会成员，微博平台资深架 构师，致力于高性能高可用互联网服务开发，及高效率团队建设。从2010年开始深度参与微博平台的建设，目前工作重心为微博服务在无线环境下 的端到端全链路优化。业余时间他是一个一岁女孩的爸爸，最擅长以45°凉开水冲泡奶粉。

原文链接：http://www.infoq.com/cn/articles/weibo-relation-service-with-redis

HDFS设计思想

作者：@HUST张友东
网站：http://blog.yunnotes.net/
攻城师@阿里巴巴，关注分布式存储

硬件故障是常态

HDFS的目标是在有机器故障时仍能保证数据可靠，并自动进行故障恢复。

流式数据访问

HDFS主要为批处理应用设计，而非交互式应用，看重数据访问的吞吐量而非访问延迟。

海量大文件存储

HDFS主要针对大文件存储设计，一个典型的HDFS文件大小在数G到数T之间，由多个64M的block组成，一个集群内能支持海量的文件（百万至千万级别）。

简单的一致性模型

HDFS提供一次写多次读（write-once-read-many)的访问模型，文件只允许在末尾追加数据，这种模型简化了多副本一致性问题，提供高的数据访问吞吐量，非常适合做map/reduce。

迁移计算vs迁移存储

【把计算调度到数据所在的位置远】比【把数据迁移到计算所在位置】成本低，HDFS提供了接口将应用移到离数据近的位置。

支持异构软硬件

HDFS基于跨平台的java语言开发，能方便的兼容各种异构的软硬件。

全局中心节点

NameNode（NN)是HDFS里的中心管理节点，NN上的元数据全内存化，元数据主要分为两个部分:(1) 文件的元信息，如大小、时间戳、包含的block等；(2) block到DataNode(DN)的映射关系。前者持久化在NN本地，而后者则是在集群里DN启动时，由DN汇报上来。

元数据持久化

NN以fsimage的形式将集群内所有文件的元数据存储到本地，成，运行过程中NN会将文件系统的更改（如创建新的文件）记录到editlog。NN启动时，加载fsimage，并合并editlog的更改日志，在内存里建立整个文件系统的目录树结构；

存储节点管理

DN在启动时，会向NN汇报存储的block信息，并周期性的向NN发送心跳信息，NN超过一定时间没有收到DN的心跳，就认为DN发生故障。

副本放置策略

HDFS通常配置3个副本，在创建block时，会将其中二个副本存储到离客户端近的机架里，将第三个副本存储到另外的机架；由于机架内部的网络带宽远大于机架间的网络带宽，HDFS的策略实际上是数据安全（所有副本分散在不同机架）与写数据效率（所有副本分散在同一机架）的折中方案。

故障恢复

当DN机器发生故障时（通常是磁盘坏掉），该磁盘上存储block的副本都会出现副本不足，NN会复制这批block，让其副本达到安全值。

安全模式

HDFS集群刚启动时，所有的DN都会向NN汇报block，当集群规模较大时汇报时间也会较长。NN启动后会先以安全模式运行一段时间（如30s），等待DN汇报block完毕，待安全模式时间过后，一旦NN检测到block副本数不足，就会对block进行复制。

负载均衡

当集群内某个DN空闲的存储空间太少时，NN会主动将该DN的一些数据（block）迁移到其他的DN上；另外，当某个block的数据访问量较大时，NN会主动创建该block额外的副本，用于分担访问请求。

数据完整性

用户从DN读取文件数据时，因DN磁盘或网络问题、软件bug等原因，客户端可能读到错误的数据。HDFS在写入文件时，为文件的每个block计算校验和，当读取文件数据时，客户端通过对数据计算校验和来确认读到数据的完整性。

流水线复制

客户端在写入时，为保证最大吞吐量，客户端会将数据先缓存到本地文件，当客户端缓存的数据超过一个block大小时，会向NN申请创建block，将数据写到block里。在往block的多个副本写入数据时，HDFS采用流水线复制的方式，比如block的三个副本在A、B、C上，客户端将数据推送给最近的A、A收到数据后立即推送给B、B推送给C，等所有副本写入成功后，向客户端返回写入成功，当文件所有的block都写入成功后，客户端向NS请求更新文件的元信息。

原文链接：http://blog.yunnotes.net/index.php/hdfs_design/
转自：Yun Notes

微博CacheService架构浅析

作者：@麦俊生
一边敲着代码一边思考着人生。。。linux C/C++, java, shell coder, debuging、高性能、分布式

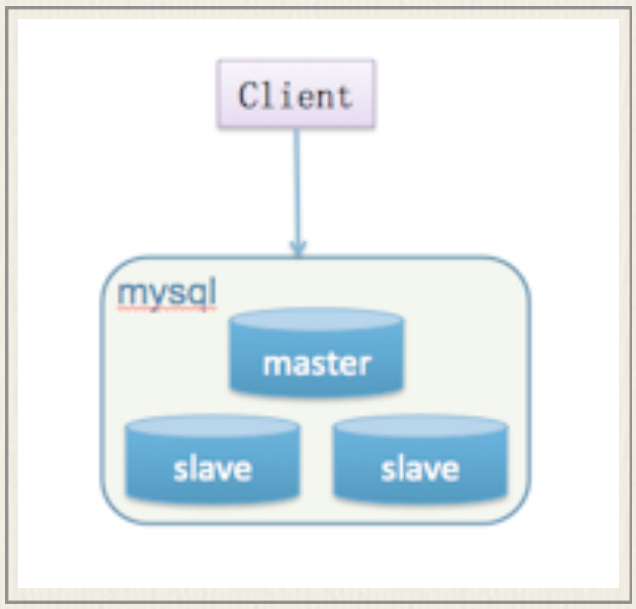
微博作为国内最大的社交媒体网站之一，每天承载着亿万用户的服务请求，这些请求的背后，需要消耗着巨大的计算、内存、网络、I/O等资源。而且因为 微博的产品特性，节假日、热门事件等可能带来突发数倍甚至十几倍的访问峰值，这些都对于支撑微博的底层基础架构提出了比较严苛的要求，需要满足：

1. 每秒数十万的用户请求
2. 数据更新的实时性
3. 服务请求的低响应时间
4. 99.99%以上的服务可用性

为了满足业务的发展需要，微博平台开发了一套高性能高可用的CacheService架构用于支撑现有线上的业务系统的运转。但“冰动三尺非一日之寒”，微博的Cache架构也是经历了从无到有，不断的演进过程。

基于MySQL的Web架构

最初的微博系统，系统的访问量都比较小，简单的基于数据库(MySQL)已经能够满足业务需求，开发也比较简单，简单的架构示意图如下：



随着微博的推广和名人用户入驻微博，带动了用户量的快速增长，访问量也与日俱增，这个时候，简单基于MySQL的架构已经略感吃力，系统响应也比较 缓慢，因为MySQL是一个持久化存储的解决方案，数据的读写都会经过磁盘，虽然MySQL也有buffer pool，但是无法根据业务的特性做到很细粒度的控制，而在微博这种业务场景下，配置了SAS盘的MySQL服务单机只能支撑几千的请求量，远小于微博的 业务请求量。

基于单层Cache+MySQL的Web架构

针对请求量增大的问题，一般有几种解决方案：

1. 业务架构改造，但是在这种场景下，这种方案的可行性不高。
2. MySQL进行从库扩容，虽然能够解决问题，但是带来的成本也会比较高，而且即使能够抗住请求量，但是资源的响应时间还是无法满足期望的结果，因为磁盘的读取的响应时间要相对比较慢，普通的15000转/分钟的SAS盘的读取延迟平均要达到2ms以上。
3. 在MySQL之上架构一层缓存，把热门请求数据缓存到Cache，基于Cache+MySQL的架构来提供服务请求。

考虑到整体的改动和成本的因素，基于方案3)比较适合微博的业务场景。而应该使用什么类型的Cache比较合适呢？

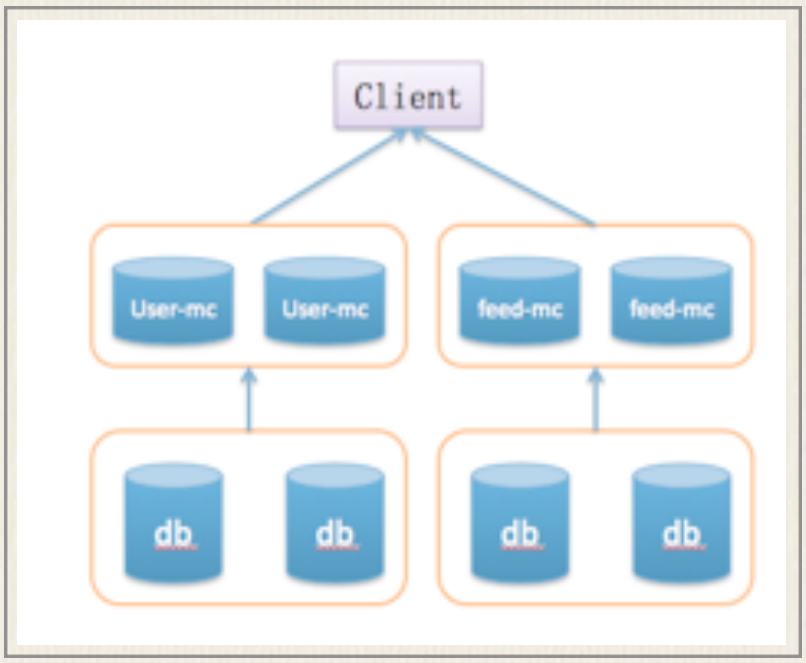
比较常见的Cache解决方案有：

1. Local Cache，通过在Web应用端内嵌一个本地的Cache，这种的优势是访问比较快，但是存在的问题也比较明显，数据更新的一致性比较难保证，因此使用的范围会有一定的限制。
2. 单机版的远程Cache，通过部署一套远程的Cache服务，然后应用端请求通过网络请求与Cache交互，为了解决应用的水平扩展和容灾问题，往往通过在client层面来实现数据的路由等。
3. 分布式的Cache，Cache服务本身是一个大集群，能够提供给各种业务应用使用，并提供了一些基本的分布式特性：水平扩展、容灾、数据一致性等等。

从系统的简单性考虑和微博场景的适用问题，最终选择了2)的方式，基于开源的Memcached来作为微博的Cache方案。

Memcached是一个分布式Cache Server，提供了key-value型数据的缓存，支持LRU、数据过期淘汰，基于Slab的方式管理内存块，提供简单的set/get/delete等操作协议，本身具备了稳定、高性能等优点，并在业界已经得到广泛的验证。它的server端本身是一个单机版，而分布式特性是基于 client端的实现来满足，通过部署多个Memcached节点，在client端基于一致性hash(或者其他hash策略)进行数据的分散路由，定 位到具体的memcached节点再进行数据的交互。当某个节点挂掉后，对该节点进行摘除，并把该节

点的请求分散到其他的节点。通过client来实现一 定程度的容灾和伸缩的能力。



这种架构经过一段时间的蜜月期后，也逐步遇到了一些问题。

- 节点挂掉导致的瞬间的峰值问题
  比如部署有5个Memcached节点，对key做一致性hash将key散落分布到5个节点上，那么如果其中有1个节点挂掉，那么这个时 候会有20%原本Cache hit的请求穿透到后端资源(比如D-B)。对于微博而言，多数核心资源的Cache hit的比例是99%，单组资源的QPS可能就达到100W以上的级别，如果这个时候有20%的穿透，那么相当于后端资源需要抗住20W以上的请求，这对 于后端资源来说，明显压力过大。

- 某组资源请求量过大导致需要过多的节点
  微博的Feed业务是Cache资源的消耗大户，几十万的QPS，GB(Byte)级别以上的带宽消耗，这个时候，至少需要十几个 Memcached节点单元才能够抗住请求，而过多的Memcached节点请求会导致multiget的性能有弱化，因为这个时候keys分散到的 Memcached节点会比较多，因此当进行拉取聚合的时候，性能会受影响，同时mutliget的响应时间受最慢的那个节点的影响，从而无法达到服务的 SLA要求。

- Cache的伸缩容和节点的替换动静太大
  对于微博这种会在热点事件、节假日等发生时会有一些变态峰值(往往是数倍或者数十倍)的场景而言，实时的动态伸缩容很是必要，而因为通过client端实例化的Memcached资源节点相对比较固定，因此要进行伸缩容需要:

  1. 进行一次代码的线上变更，进行节点配置的变更，而如果依赖该某组资源的应用系统比较多，比如底层的认证资源，那么需要对多个

业务系统变更，这一动静不可谓不小，特别是遇到紧急情况，这个会导致操作的执行很缓慢。

2. 需要解决读写导致的一致性问题，假如有一些业务系统在读取Cache，有一些业务系统在写入Cache，而正常的变更是比较难让这些系统在某一刻全部执行节点的配置切换。
3. 需要使用新的节点替换老的节点(比如更换物理机)，面临和上面类似的问题。

- 过多资源带来的运维问题

Cache资源组是按业务去申请，当业务特别多的时候，Cache资源组也会很多，这个时候要对这些资源进行运维管理如调整，将会变得不容易。而且随着时间的演进，一些比较古老的资源年老失修的情况，要进行运维调整就更为不容易。

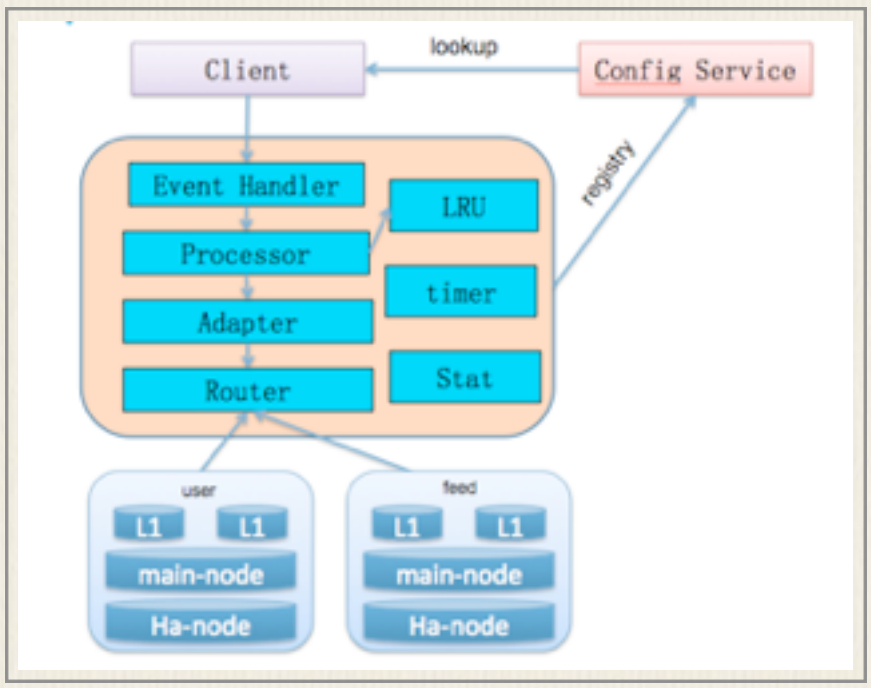
- Cache架构要用得好的复杂度

会用和用得好是两个不同概念。如果Cache架构需要每个业务开发很熟练才能够用得好，而不会因为Cache的不当使用而导致线上服务出现稳定性问题、以及成本的浪费等各种问题的话，这种对于需要陆续补进新人的团队现状而言，出问题将会是一种常态。 因此要解决这种问题，那么需要提供一种足够简单的Cache使用方式给业务应用方，简单到只有set/get/delete等基本命令的操作，而无需要他 们关心底层的任何细节。

分布式CacheService架构

为了解决这些问题，微博的Cache服务架构进行了演进，通过把Cache服务化，提供一个分布式的CacheService架构，简化业务开发方的使用，实现系统的动态伸缩容、容灾、多层Cache等相关功能。

CacheService架构示意图如下：



系统由几个模块组成：

- ConfigService
  这一模块是基于现有微博的配置服务中心，它主要是管理静态配置和动态命名服务的一个远程服务，能够在配置发生变更的时候实时通知监听的config client。

- proxy层
  这一模块是作为独立的应用对外提供代理服务，用来接收来自业务端的请求，并基于路由规则转发到后端的Cache资源，它本身是无状态的节点。它包含了如下部分：

  - 异步事件处理(event handler): 用来管理连接、接收数据请求、回写响应。
  - Processor: 用来对请求的数据进行解析和处理。
  - Adapter：用来对底层协议进行适配，比如支持MC协议，Redis协议。
  - Router: 用来对请求进行路由分发，分发到对应的Cache资源池，进而隔离不同业务。
  - LRU Cache: 用来优化性能，缓解因为经过proxy多一跳(网络请求)而带来的性能弱化。
  - Timer: 用来执行一些后端的任务，包含对底层Cache资源健康状态的探测等。

- Proxy启动后会去从config Service加载后端Cache资源的配置列表进行初始化，并接收configService的配置变更的实时通知。

- Cache资源池
  这一模块是作为实际数据缓存的模块，通过多层结构来满足服务的高可用。 其中Main-node是主缓存节点，Ha-Node是备份节点，当Main-node挂掉后，数据还能够从Ha-Node节点获取避免穿透到后端资 源，L1-node主要用来抗住热点的访问，它的容量一般比Main-node要小，其中L1-node可支持多组，方便进行水平扩容以支撑更高的吞吐。

- Client客户端
  这一模块主要是提供给业务开发方使用的client(sdk包)，对外屏蔽掉了所有细节，只提供了最简单的get/set/delete等协议接口，从而简化了业务开发方的使用。
  应用启动时，Client基于namespace从configService中获取相应的proxy节点列表，并建立与后端proxy的 连接。正常一个协议处理，比如set命令，client会基于负载均衡策略挑选当前最小负载的proxy节点，发起set请求，并接收proxy的响应返 回给业务调用端。

Client会识别configService推送的proxy节点变更的情况重建proxy连接列表，同时client端也会做一些容灾，在proxy节点出现问题的时候，把proxy进行摘除，并定期探测是否恢复。

目前微博平台部分业务子系统的Cache服务已经迁移到了CacheService之上，它在实际的运行过程中也取得了良好的性能表现，目前整个集群在线上每天支撑着超过300W的QPS，平均响应耗时低于1ms。

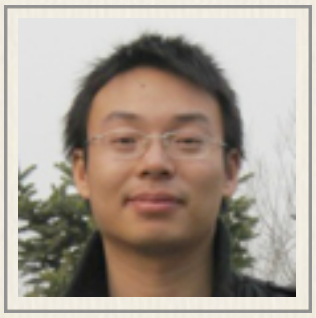
它本身具备了以下特性:

- 高可用保证
  所有的数据写入请求，CacheService会把数据双写到ha的节点，这样，在main-node挂掉的时候，会从ha-node读取数据，从而防止节点fail的时候给后端资源(DB等)带来过大的压力。
- 服务的水平扩展
  CacheService proxy节点本身是无状态的，在proxy集群存在性能问题的时候，能够简单的通过增减节点来伸缩容。而对于后端的Cache资源，通过增减L1层的 Cache资源组，来分摊对于main-node的请求压力。这样多数热点数据的请求都会落L1层，而L1层可以方便的通过增减Cache资源组来进行伸 缩容。
- 实时的运维变更
  通过整合内部的config Serv-
  ice系统，能够在秒级别做到资源的扩容、节点的替换等相关的运维变更。
- 跨机房特性:
  微博系统会进行多机房部署，跨机房的服务器网络时延和丢包率要远高于同机房，比如微博广州机房到北京机房需要40ms以上的时延。
  CacheService进行了跨机房部署，对于Cache的查询请求会采用就近访问的原则，对于Cache的更新请求支持多机房的同步更新。

目前微博的分布式CacheService架构在简化了业务开发使用的同时，提高了系统的可运维性和可用性。接下来的架构的改造方向是提供后端Cache资源的低成本解决方案，从单机的存储容量和单机的极限性能层面不断优化。因为对于微博的业务场景，冷热数据相对比较明显，同时长尾数据请求的比 例也不小，因而如果减少了Cache的容量，那么会导致后端资源无法抗住请求，而扩大Cache的容量，又会导致成本的浪费。而全内存的解决方案相比而言 成本相对比较高，所以热数据存放到内存，基于LRU的策略把冷数据交换到固体硬盘(SSD)，这是一种可能选择的方向。

原文链接：http://www.infoq.com/cn/articles/weibo-cacheservice-architecture

使用Etag增强iOS的URL缓存功能 - iOS移动开发周报



作者：@唐巧_boy
InfoQ编辑，Blogger，iOS开发，创业者，前网易员工。微信公共账号iOSDevTips创建者。

教程

1. 《SDWebImage缓存图片的机制》： 文章详细介绍了SDWebImage对于图片的缓存逻辑的实现细节。
2. 《使用Etag增强iOS的URL缓存功能》： iOS 下对于缓存的支持有很多，比如Core Data，它可以很方便的建模和操作本地SQLite数据库，NSUserDefaults则可以用来缓存一些用户配置等等。本篇主要谈谈使用Etag标签来解决iOS下的URL缓存问题。服务端使用的是 Tornado，原生支持Etag，而且默认处于开启状态，因此在应用层面基本不需要额外的代码处理。客户端使用NSURLCache配合 AFNetwork-ing进行网络请求。
3. 《初始化阶段 —— load 和 initialize》： 文章介绍了load函数和initialize函数各自的加载顺序。最后文章总结出：将针对于类修改放在intialize中，将针对Category的 修改放在load中。假如是修改系统的类，一般会通过添加Category来添加功能，如果修改initialize会导致原生的intialize不会 执行，所以放在load中会比较妥当。
4. 《ObjC @interface的设计哲学与设计技巧》： 学习Objective-C语言时，尤其是先学过其他编程语言再来看Objective-C时，总会对Objective-C的类声明的关键字 interface感到有点奇怪，在其它面向对象的语言中通常由class关键字来表示，而interface在Java中表示的却大约相当于 Objective-C的protocol，这个关键字的区别究竟代表了Objective-C语言的设计者怎样的思想呢，在Objective-C类设 计中需要注意哪些问题呢？作者在文章中对这个问题进行一些思考和探究。

5. 《从Facebook看移动开发的发展》：作者从Facebook的故事切入，分享了未来移动开发快速发展中，给设计师和程序员带来的挑战。
6. 《CoreData Object 变成 Fault 的另一种方式》：CoreData 是一个架构庞大、学习曲线比较陡峭的 iOS 组件，每次遇到问题都会对其有新的认识。文章分享了关于错误认知 Object（NSManagedObject）与 Context（NSManagedContext）的引用关系而导致的 Fault 问题。
7. 《关于 Mac 右键菜单》：Mac OS X Lion 的 Launch Service 用于关联应用程序和文件并维护最近打开的项目列表。在文件关联和右键菜单方面，每当系统安装一个新的应用程序，都会调用 Launch Service 的 API 注册关联的文件类型。文章详细介绍了这其中的过程。

工具

1. fnd.io: fnd.io 是一个App Store的网页版，搜索速度非常快，可以用来代替itunes进行应用的检索。
2. Shortcut Foo：Shortcut Foo是一个训练你记忆快捷键的网站，除了包括iOS的第三方IDE AppCode的内容外，还包括vim，emacs, git等内容。
3. FuzzyAutocomplete：FuzzyAutocomplete是一个Xcode自动补全插件，刚刚更新到了2.0版本。它可以让你不需要再遵循从头匹配的原则来补全代码，而是随便输入你记得的关键字来进行匹配，整个插件的响应速度也非常快。

开源项目

1. Lockbox: Lockbox是一个帮助你方便地将数据保存到keychain中的开源工具类。
2. Framework7： Framework7 是一个功能齐全的 HTML 框架，用来构建 iOS7 应用程序。Framework7 允许你使用Javascript代码来实现应用的列表，侧边栏，弹出框等控件。

原文链接：http://www.infoq.com/cn/news/2014/04/etag-improve-ios-url-function

使用objection来模块化开发iOS项目

作者：@lzyy

技术路线：ActionScript / HTML+CSS+Javscript -> PHP -> Python -> Objective-C

objection 是一个轻量级的依赖注入框架，受Guice的启发，Google Wallet 也是使用的该项目。「依赖注入」是面向对象编程的一种设计模式，用来减少代码之间的耦合度。通常基于接口来实现，也就是说不需要new一个对象，而是通过相关的控制器来获取对象。2013年最火的PHP框架 laravel 就是其中的典型。

假设有以下场景：ViewControllerA.view里有一个button，点击之后push一个ViewControllerB，最简单的写法类似这样：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    UIButton *button = [UIButton buttonWithType:UIButtonTypeSystem];
    button.frame = CGRectMake(100, 100, 100, 30);
    [button setTitle:@"Button" forState:UIControlStateNormal];
    [button addTarget:self action:@selector(buttonTapped)
      forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:button];
}

- (void)buttonTapped
{
    ViewControllerB *vc = [[ViewControllerB alloc] init];
    [self.navigationController pushViewController:vc animated:YES];
}
```

这样写的一个问题是，ViewControllerA需要import ViewControllerB，也就是对ViewControllerB产生了依赖。依赖的东西越多，维护起来就越麻烦，也容易出现循环依赖的问题，而objection正好可以处理这些问题。

实现方法是：先定义一个协议(protocol)，然后通过objection来注册这个协议对应的class，需要的时候，可以获取该协议对应的 object。对于使用方无需关心到底使用的是哪个Class，反正该有的方法、属性都有了(在协议中

指定)。这样就去除了对某个特定Class的依赖。 也就是通常所说的「面向接口编程」。

```
JSObjectionInjector *injector = [JSObjection defaultInjector]; // [1]
UIViewController <ViewControllerAProtocol> *vc = [injector
getObject:@protocol(ViewControllerAProtocol)]; // [2]
vc.backgroundColor = [UIColor lightGrayColor]; // [3]
UINavigationController *nc = [[UINavigationController alloc]
initWithRootViewController:vc];
self.window.rootViewController = nc;
```

- [1] 获取默认的injector，这个injector已经注册过ViewControllerAProtocol了。
- [2] 获取ViewControllerAProtocol对应的Object。
- [3] 拿到VC后，设置它的某些属性，比如这里的backgroundColor，因为在ViewControllerAProtocol里有定义这个属性，所以不会有warning。

可以看到这里没有引用ViewControllerA。再来看看这个ViewControllerAProtocol是如何注册到injector中的，这里涉及到了Module，对Protocol的注册都是在Module中完成的。Module只要继承JSObjectionModule这个Class即可。

```
@interface ViewControllerAModule : JSObjectionModule
@end

@implementation ViewControllerAModule
- (void)configure
{
    [self bindClass:[ViewControllerA class]
toProtocol:@protocol(ViewControllerAProtocol)];
}
@end
```

绑定操作是在configure方法里进行的，这个方法在被添加到injector里时会被自动触发。

```
JSObjectionInjector *injector = [JSObjection defaultInjector]; // [1]
injector = injector ? : [JSObjection createInjector]; // [2]
injector = [injector withModule:[[ViewControllerAModule alloc]
init]]; // [3]
[JSObjection setDefaultInjector:injector]; // [4]
```

- [1] 获取默认的 injector

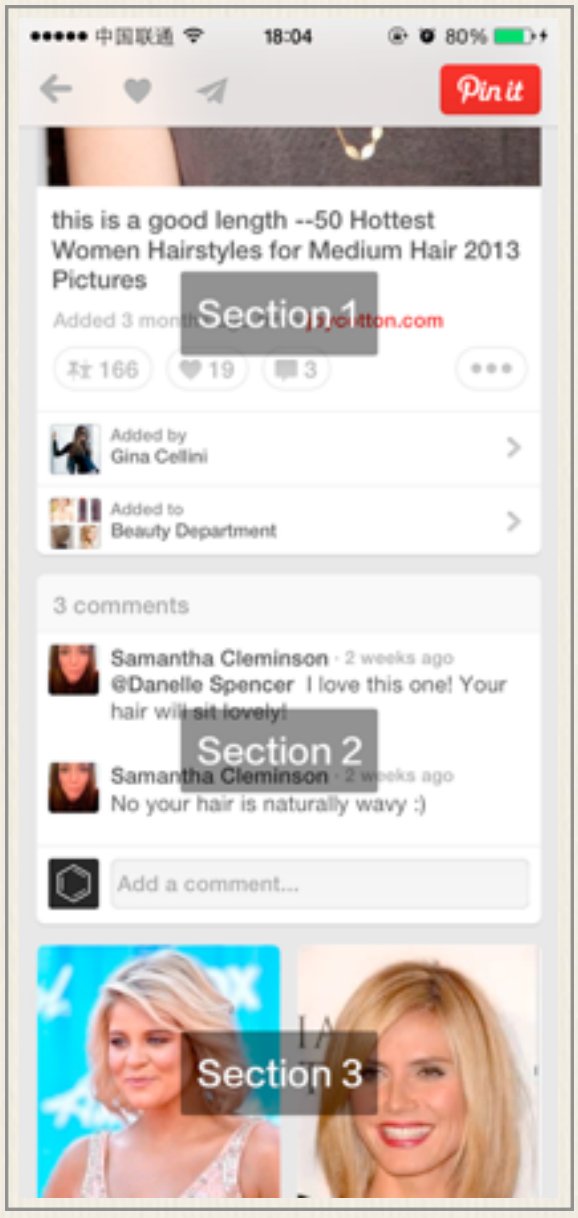
- [2] 如果默认的 injector 不存在，就新建一个
- [3] 往这个 injector 里注册我们的 Module
- [4] 设置该 injector 为默认的 injector

这段代码可以直接放到 + (void)load里执行，这样就可以避免在AppDelegate里import各种Module。

因为我们无法直接获得对应的Class，所以必须要在协议里定义好对外暴露的方法和属性，然后该Class也要实现该协议。

```
@protocol ViewControllerAProtocol <NSObject>
@property (nonatomic) NSUInteger currentIndex;
@property (nonatomic) UIColor *backgroundColor;
@end

@interface ViewControllerA : UIViewController <ViewControllerAProto-
col>
@end
```

通过objection实现依赖注入后，就能更好地实现SRP(Single Responsibility Principle)，代码更简洁，心情更舒畅，生活更美好。拿Pinterest来说，下面的页面就可以划分为3个Section。



各个Sec-
tion可以由不同的人负责，然后串到一起就行，也能一定程度地避免MVC(Mess View Controller)的出现。

总体来说，这个lib还是挺靠谱的，已经维护了两年多，也有一些项目在用，对于提高开发成员的效率也会有不少的帮助，可以考虑尝试下。

原文链接：http://blog.leezhong.com/tech/2014/04/15/use-objection-to-decouple-ios-project.html

剖析Disruptor:为什么会这么快？(一)锁的缺点

译者：@NICK

汇丰银行开发工程师

Martin Fowler写了一篇非常好的文章，里面不仅提到了Disruptor,而且还解释了Disruptor 如何应用在LMAX的架构里。里面有提及了一些目前没有涉及的概念，但最经常问到的问题是 “Disruptor究竟是什么？＂。

目前我正准备在回答这个问题，但首先回答＂为什么它会这么快？＂

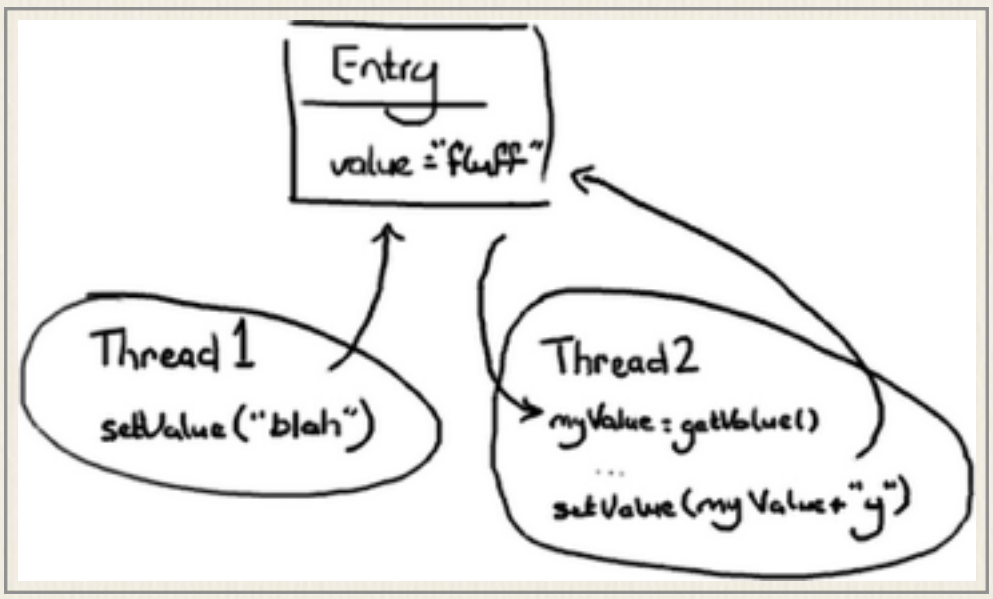
这些问题持续出现，但是我不能没有说它是干什么的就说它为什么会这么快，不能没有说它为什么这样做就说它是什么东西。

所以我陷入了一个僵局，一个如何写博客的僵局。

要打破这个僵局，我准备以最简单方式回答第一个问题，如果幸运的话，在以后博文里，如果需要解释的话我会重新提回：Disruptor提供了一种线程之间信息交换的方式。

作为一个开发者,因为＂线程＂一词的出现，我的警钟已经敲响，它意味着并发，而并发是困难的。

并发 01



想象有两个线程尝试修改同一个变量value：

情况一：线程 1 先到达

1. 变量value的值变为”blah”。
2. 然后当线程 2 到达时，变量value的值变为”blahy”。

情况二：线程 2 先到达

1. 变量value的值变为”fluffy”。
2. 然后当线程 1 到达时，值变为”blah”。

情况三：线程 1 与线程 2 交互

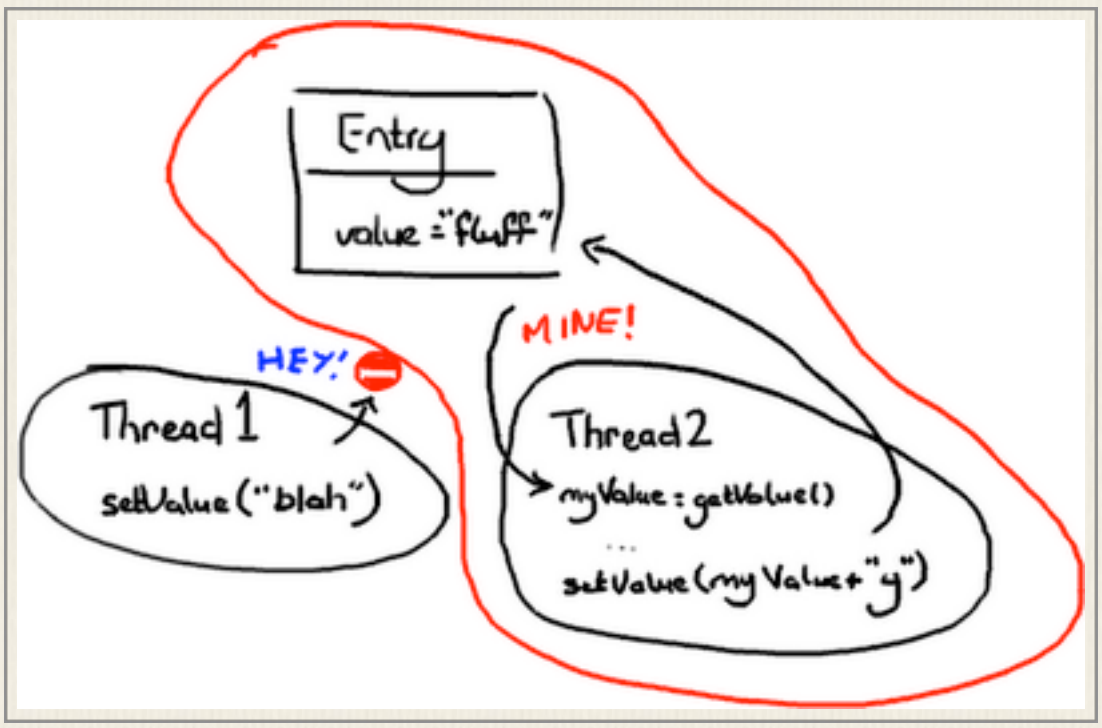
1. 线程 2 得到值 " fluff " 然后赋给本地变量myValue。
2. 线程 1 改变value的值为”blah”。
3. 然后线程 2 醒来并把变量value的值改为”fluffy”

情况三显然是唯一一个是错误的，除非除非你认为wiki编辑的幼稚做法是正确的（Google Code Wiki，我一直在关注你）。其他两种情况主要是看你的意图和想要达到的效果。线程 2 可能不会关心变量value的值是什么，主要的意图就是在后面加上字符 ‘y´ 而不管它原来的值是什么，在这种前提下，情况一和情况二都是正确的。

但是如果线程 2 只是想把 " fluff " 改为”fluffy”，那么情况二和三都不正确。

假定线程 2 想把值设为”fluffy”,有几种办法可以解决这个问题：

办法一：悲观锁



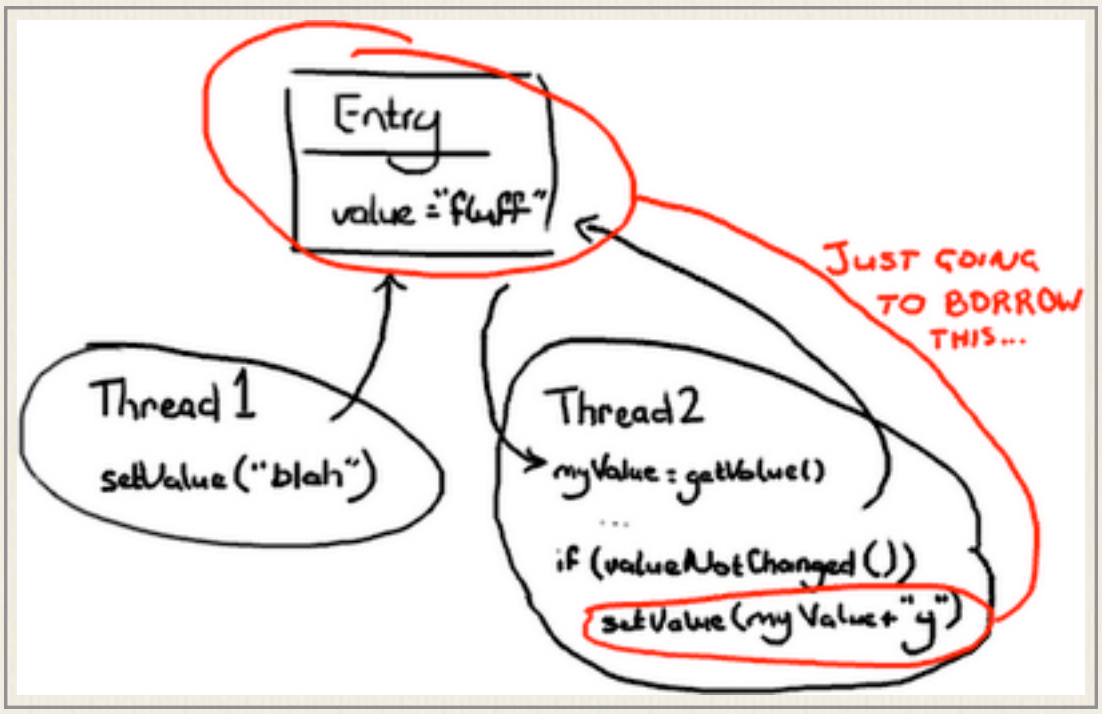
(“No Entry”的标志对于在没有在英国开车的人看得明白不？)

悲观锁和乐观锁这两个词通常在我们谈论数据库读写时经常会用到，但原理可以应用到在获得一个对象的锁的情况。

只要线程 2 一获得Entry 的互斥锁，它就会阻击其它线程去改变它，然后它就可以随意做它要做的事情，设置值，然后做其它事情。

你可以想象这里非常耗性能的，因为其它线程在系统各处徘徊着准备要获得锁然后又阻塞。线程越多，系统的响应性就会越慢.

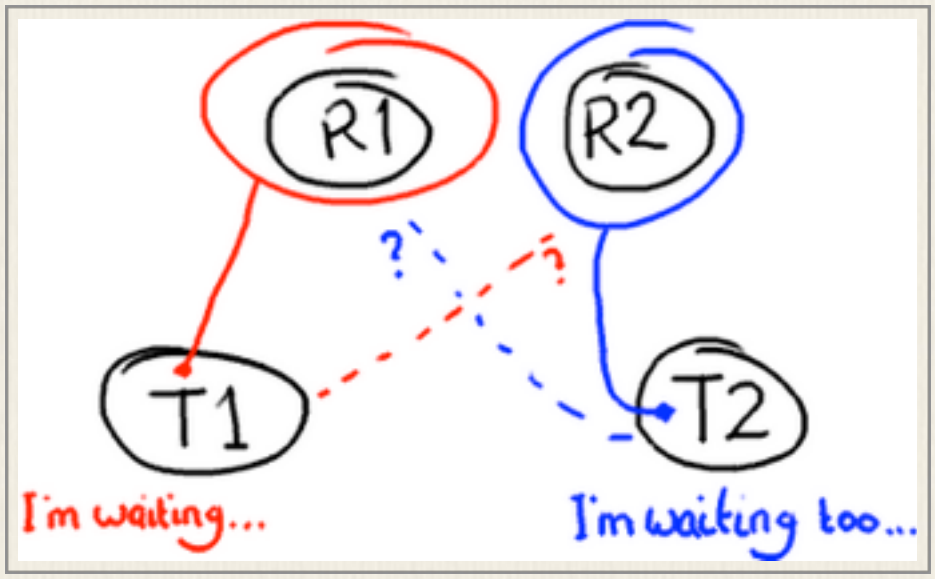
办法二：乐观锁



在这种情况，当线程 2 需要去写Entry时才会去锁定它．它需要检查Entry自从上次读过后是否已经被改过了。如果线程 1 在线程 2 读完后到达并把 值改为”blah”,线程 2 读到了这个新值，线程 2 不会把 " fluffy " 写到Entry里并把线程 1 所写的数据覆盖．线程 2 会重试（重新读新的值，与旧 值比较，如果相等则在变量的值后面附上’y’）,这里在线程 2 不会关心新的值是什么的情况．或者线程 2 会抛出一个异常，或者会返回一个某些字段已更新的标志，这是在期望把”fluff”改为”fluffy”的情况．举一个第二种情况的例子，如果你和另外一个用户同时更新一个Wiki的页面，你会告诉另外一 个用户的线程 Thread 2，它们需要重新加载从Thread1来新的变化，然后再提交它们的内容。

潜在的问题：死锁

锁定会带来各种各样的问题，比如死锁，想象有 2 个线程需要访问两个资源



如果你滥用锁技术，两个锁都在获得锁的情况下尝试去获得另外一个锁，那就是你应该重启你的电脑的时候了。（校注：作者还挺幽默）

很明确的一个问题：锁技术是慢的．．

关于锁就是它们需要操作系统去做裁定。线程就像两姐妹在为一个玩具在争吵，然后操作系统就是能决定他们谁能拿到玩具的父母，就像当你跑向你父亲告诉 他你的姐姐在你玩着的时候抢走了你的变形金刚 – 他还有比你们争吵更大的事情去担心，他或许在解决你们争吵之前要启动洗碗机并把它摆在洗衣房里。如果你把你 的注意力放在锁上，不仅要花时间来让操作系统来裁定。Disruptor论文中讲述了我们所做的一个实验。这个测试程序调用了一个函数，该函数会对一个 64位的计数器循环自增5亿次。当单线程无锁时，程序耗时300ms。如果增加一个锁（仍是单线程、没有竞争、仅仅增加锁），程序需要耗时 10000ms，慢了两个数量级。更令人吃惊的是，如果增加一个线程（简单从逻辑上想，应该比单线程加锁快一倍），耗时 224000ms。使用两个线程对 计数器自增5亿次比使用无锁单线程慢1000倍。**并发很难而锁的性能糟糕**。我仅仅是揭示了问题的表面，而且，这个例子很简单。但重点是，如果代码在多线程环境中执行，作为开发者将会遇到更多的困难：

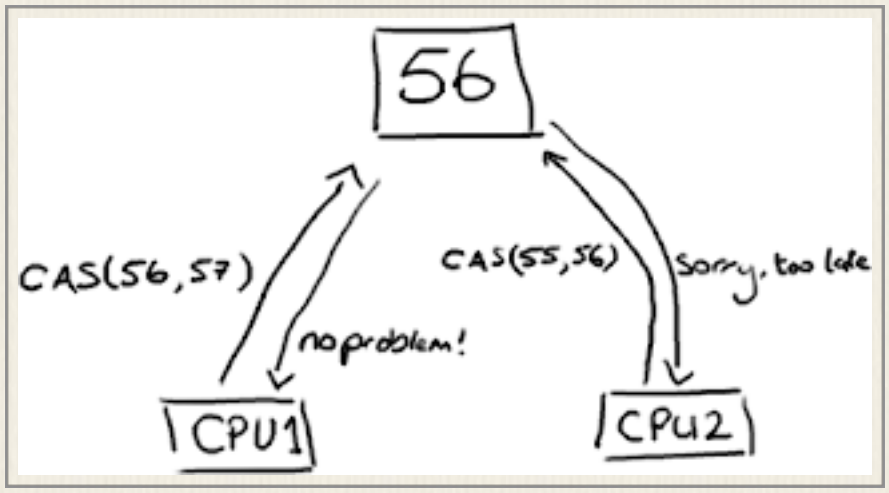
- 代码没有按设想的顺序执行。上面的场景3表明，如果没有注意到多线程访问和写入相同的数据，事情可能会很糟糕。
- 减慢系统的速度。场景3中，使用锁保护代码可能导致诸如死锁或者效率问题。

这就是为什么许多公司在面试时会多少问些并发问题（当然针对Java面试）。不幸的是，即使未能理解问题的本质或没有问题的解决方案，也很容易学会如何回答这些问题。

Disruptor如何解决这些问题。

首先，Disruptor根本就不用锁。

取而代之的是，在需要确保操作是线程安全的（特别是，在多生产者的环境下，更新下一个可用的序列号）地方，我们使用CAS（Compare And Swap/Set）操作。这是一个CPU级别的指令，在我的意识中，它的工作方式有点像乐观锁——CPU去更新一个值，但如果想改的值不再是原来的值，操作就失败，因为很明显，有其它操作先改变了这个值。



注意，这可以是CPU的两个不同的核心，但不会是两个独立的CPU。

CAS操作比锁消耗资源少的多，因为它们不牵涉操作系统，它们直接在CPU上操作。但它们并非没有代价——在上面的试验中，单线程无锁耗时300ms，单线程有锁耗时10000ms，单线程使用CAS耗时5700ms。所以它比使用锁耗时少，但比不需要考虑竞争的单线程耗时多。

回到Disruptor，在我讲生产者时讲过ClaimStrategy。 在这些代码中，你可以看见两个策略，一个是SingleThreadedStrategy（单线程策略）另一个是MultiThreadedStrategy（多线程策略）。你可能会有疑问，为什么在只有单个生产者时不用多线程的那个策略？它是否能够处理这种场景？ 当然可以。但多线程的那个使用了AtomicLong（Java提供的CAS操作），而单线程的使用long，没有锁也没有CAS。这意味着单线程版本会非常快，因为它只有一个生产者，不会产生序号上的冲突。

我知道，你可能在想：把一个数字转成AtomicLong不可能是Disruptor速度快的唯一秘密。当然，它不是，否则它不可能称为“为什么这么快（第一部分）”。

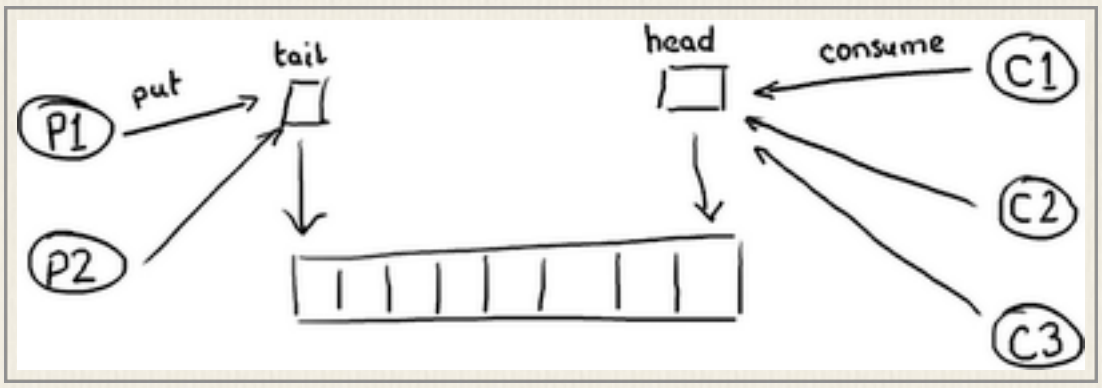
但这是非常重要的一点——在整个复杂的框架中，只有这一个地方出现多线程竞争修改同一个变量值。这就是秘密。还记得所有的访问对象都拥有序号吗？如 果只有一个生产者，那么系统中的每一个序列号只会由一个线程写入。这意味着没有竞争、不需要锁、甚至不需要CAS。在ClaimStrategy

中，如果 存在多个生产者，唯一会被多线程竞争写入的序号就是 Claim-Strategy 对象里的那个。

这也是为什么Entry中的每一个变量都只能被一个消费者写。它确保了没有写竞争，因此不需要锁或者CAS。

回到为什么队列不能胜任这个工作

因此你可能会有疑问，为什么队列底层用Ring-Buffer来实现，仍然在性能上无法与 Disruptor 相比。队列和最简单的ring buffer只有两个指针——一个指向队列的头，一个指向队尾：

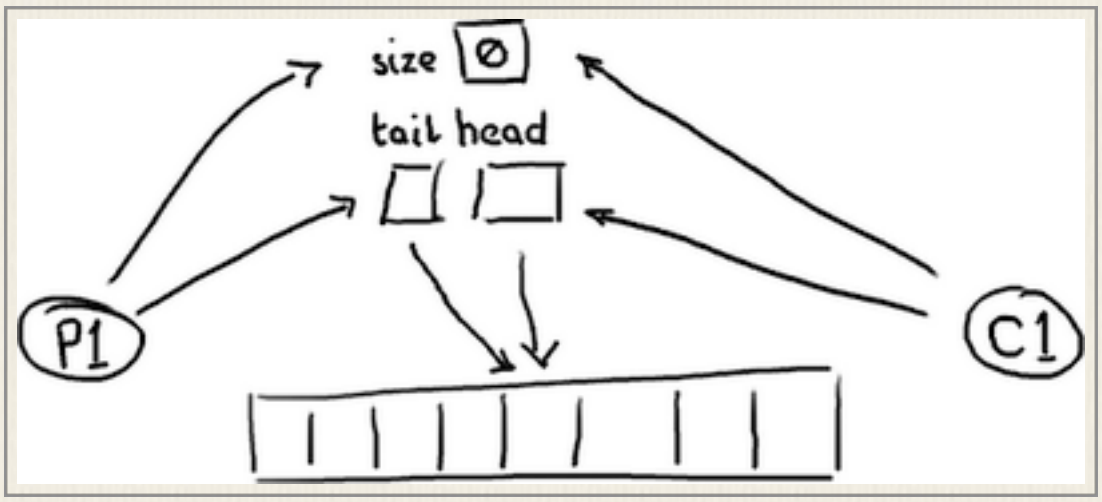


如果有超过一个生产者想要往队列里放东西，尾指针就将成为一个冲突点，因为有多个线程要更新它。如果有多个消费者，那么头指针就会产生竞争，因为元素被消费之后，需要更新指针，所以不仅有读操作还有写操作了。

等等，我听到你喊冤了！因为我们已经知道这些了，所以队列常常是单生产者和单消费者（或者至少在我们的测试里是）。

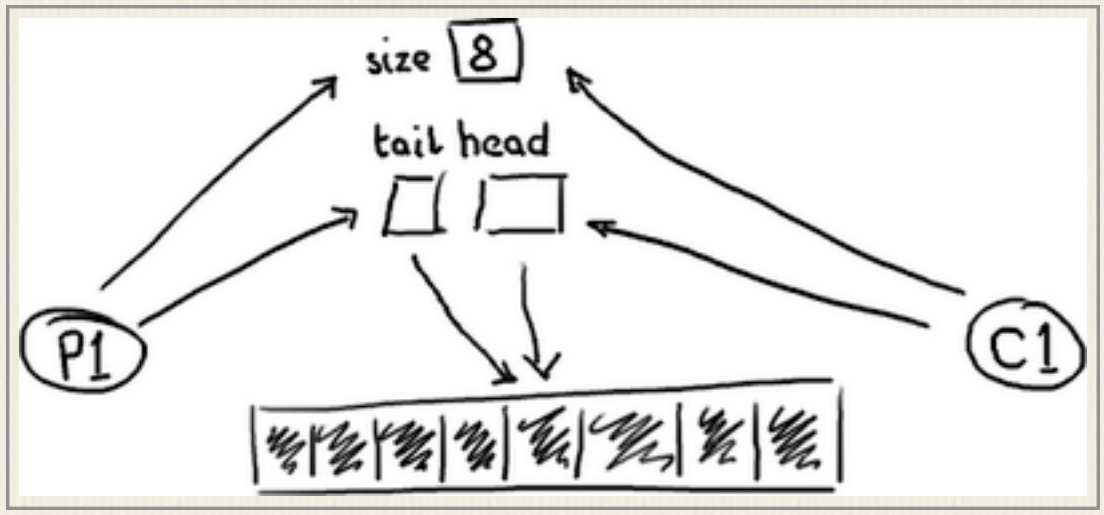
队列的目的就是为生产者和消费者提供一个地方存放要交互的数据，帮助缓冲它们之间传递的消息。这意味着缓冲常常是满的（生产者比消费者快）或者空的（消费者比生产者快）。生产者和消费者能够步调一致的情况非常少见。

所以，这就是事情的真面目。一个空的队列：



…

一个满的队列：



(校对注：这应该是一个双向队列)
队列需要保存一个关于大小的变量，以便区分队列是空还是满。否则，它需要根据队列中的元素的内容来判断，这样的话，消费一个节点（Entry）后需 要做一次写入来清除标记，或者标记节点已经被消费过了。无论采用何种方式实现，在头、尾和大小变量上总是会有很多竞争，或者如果消费操作移除元素时需要使 用一个写操作，那元素本身也包含竞争。

基于以上，这三个变量常常在一个cache line里面，有可能导致false sharing。因此，不仅要担心生产者和消费者同时写size变量（或者元素），还要注意由于头指针尾指针在同一位置，当头指针更新时，更新尾指针会导致缓存不命中。这篇文章已经很长了，所以我就不再详述细节了。

这就是我们所说的“分离竞争点问题”或者队列的“合并竞争点问题”。通过将所有的东西都赋予私有的序列号，并且只允许一个消费者写Entry对象中的变量来消除竞争，Disruptor 唯一需要处理访问冲突的地方，是多个生产者写入 Ring Buffer 的场景。

总结

Disruptor相对于传统方式的优点：

1. 没有竞争=没有锁=非常快。
2. 所有访问者都记录自己的序号的实现方式，允许多个生产者与多个消费者共享相同的数据结构。
3. 在每个对象中都能跟踪序列号（ring buffer，claim Strategy，生产者和消费者），加上神奇的cache line padding，就意味着没有为伪共享和非预期的竞争。

英文原文：http://mechanitis.blogspot.com/2011/07/dissecting-disruptor-why-its-so-fast.html
本文链接：http://ifeve.com/locks-are-bad/

LinkedIn高级分析师王益：大数据时代的理想主义和现实主义

作者：李盼（来自图灵访谈）

对话国外知名技术作者 讲述码农精彩人生

你听得见他们，他们也听得见你

王益，LinkedIn高级分析师。他曾 在腾讯担任广告算法和策略的技术总监，在此期间他发明了并行机器学习系统“孔雀”，它可以从数十亿的用户行为或文本数据中学习到上百万的潜在主题，该系统 被应用在腾讯可计算广告业务中。在此之前，他在Google担任软件工程师，并开发了一个分布式机器学习工具，这个工具让他获得了2008年的 “Google APAC 创新奖”。王益曾在清华大学和香港城市大学学习，并取得了清华大学机器学习和人工智能的博士学位。此外，他还是IEEE的高级会员，著有《推荐系统实践》。

不认输

“再想想既然中学时能自学大学课程，当下好歹也该试着突破一下困境吧。于是从高中数学课本开始看，一直看到机器学习专业的教材。”

你从什么时候开始编程的?

我看别人编程很早。自己动手是在小学五年级，那时候爸爸买了一台中华学习机，也就是中科院对Apple II的克隆。长大之后才听说当时台湾宏碁电脑公司也克隆了Apple II，取名叫“小教授”。当时我的四叔有一台“笔记本”电脑，没有显示器，但是集成了一个肥皂盒那么大的打印机——每输入一行指令，就在纸带上打印出来。 这些都让我好奇和着迷。

我迷恋编程是从初三毕业后的那个暑假：用6502汇编语言和BASIC语言混合写了一个在电视上显示的“打猎”游戏。其实那时候386都已经大行其 道了，而家里的电脑一直没有更新。主要原因是妈妈担心我用电脑玩游戏。高一的寒假，在邻居易金务伯伯（国防科技大学人文与社会科学学院教授）劝说下，爸妈 给我买了一台486。

其实体会过编程的乐趣的人，不容易沉迷于游戏 ——因为前者是人设计规则，让机器照着做；后者是人跟着机器的规则动，有点儿像围栏里的牛一样——当然是前者更有意思。我从高一开始接触和自学C++语 言。在高中阶段经常逃课，跑回家写程序。好几位老师很担心，多次来家访。我也很惭愧，但是抵不住编程的诱惑。

我高二的时候自学完了计算机本科专业课程，通过了“程序员”和“高级程序员”认证考试。这个经历锻炼了我的自学能力，培养了自信，逐渐摆脱了“不如邻居家孩子成绩好”的心理压力。

大学学的是什么专业?

我在国防科技大学读的本科，计算机专业。这里是银河和天河系列超级计算机的家——每一代机器都是当时的世界顶级水平；最近一次是2013年6月天河2号夺魁世界第一超级计算机。新生入校的思想教育就是参观这些机器，绝对让人振奋。

和其他学校相比，国防科大计算机系的软硬件教育都很严格。比如本科计算机原理课程的大作业通常是用集成电路组装一台计算机，但是在国防科大基

本没有集成电路（只有一片很原始的8位ALU）。**换句话说，需要学生自己设计CPU**（包括指令集），并且用最基本的电路元器件实现出来。而那些电阻电容三极管故障率很高，所以构造电脑的过程中要严格测试，规划回避风险。

另一个例子是编译原理的大作业，清华课程要求是“修改PL/1语言的编译器，增加一种语法”，国防科大的要求是“设计一种语言并且实现其编译器”。当然清华的课程设置是有道理的——让同学们用最合理的精力和时间付出取得最大的锻炼。**而国防科大的课程设置的目的是——顶级计算技术的薪火相传**。给本科生上课的老师有很多是从1960年代就走在计算机研究最前沿的老教授。

这里有我的恩师李思昆教授——三界“银河功臣”、文职一级（相当于武职的中将）。国防科大有一个“优异生”制度——选择基础好的本科生当研究生培养。我从大二开始成为优异生，进了教研室跟李老师学习计算机图形学（computer graphics）。这比初中时那个“打猎”游戏有意思多了。当时我开发了一个浏览器插件，可以在网页里嵌入可编程的三维图形效果，并且可编程可配置性比 当时各种VRML插件更高。**可惜当时不知道怎么产业化，要不然说不定可以和后来人尽皆知的可编程二维图形技术Flash较量一下。**

在大学的学习过程中有什么有趣的学习经历吗?

因为中学时已经自学了本科专业课程，所以那时我经常借口“教研室有任务”逃课，一天到晚泡在实验室里写程序。长沙夏天的傍晚常有暴雨。有一次我专注编程没关窗，直到飘进来的雨点把屏幕和眼镜都打湿到看不清了，才意识到。

国防科大是一所军校，体育课都是军体项目：投手榴弹、5000米跑、三级跳、单双杠三动作到六动作。我小时候体弱多病，一开始不适应。我的同学们给 了我很多帮助，后来还把我拉进了我们学员队的排球队做候补。我们学员队长陈传宝（江湖人称“宝哥”）是八一体工队的专业运动员，也给了我很多鼓励。感谢他 们帮我养成了锻炼身体的习惯。**现在公司和家之间有一条biking trail，我经常下午抽一个小时在上面跑8公里**；另外每天骑自行车上下班，往返共一个小时。

你在香港城市大学和清华大学都从事过机器学习方面的学习和研究，你觉得两所大学的学术环境和风格有什么不同？你在两所大学的收获是什么？

香港的大学的发展历程更像欧美大学，大陆大学的建设深受苏联影响，这是主要区别。前人之述备矣。对我个人而言，清华百年老校，有温和敦厚的长者气质；城大始建于上世纪80年代，有青春活泼的氛围。这可以从我的一段经历讲起：

我从小数学不好。在计算机领域选了图形学，是因为我以为这里数学简单。我的博士导师周立柱老师是数据库方面的专家，但是他“因材施教”，推荐我去微 软亚洲研究院图形学组实习。但是和研究院汇聚的全国高校精英同学们比起来，我脑子反应慢，研究工作做的不够好，所以跟周老师又要了一个去城市大学学“有道 理的”机器学习的机会。**不料去了以后才发现身边的同学都是数学“童子功”，他们嘴里蹦出来的词儿我都闻所未闻**。午饭时大家顺便聊点儿科学问题，我完全听不懂。于是陷入深深的自卑感里了，没有勇气面对困难，每天混吃等死。

我在城大的导师刘志强教授一生经历过很多风浪，为人刚毅果敢。见我一副不可救药的样子，于是决断“你这样不能白拿每个月一万多港元的助研工资，你还 是回去吧。”但是实际上他和周老师商量，把我放在清华深圳研究生院，托付给当时任信息学部主任的钟玉琢老师照顾，他俩每隔一段时间来深圳看我。可我那时候 不知道这些，心理压力叫一个大。虽然自由选择研究方向，可是在微软和香港都没有做出成绩。博士读到第四年，一篇论文也没发表过。一时间心灰意冷，考虑辍 学。

可是认输又觉得对不起周老师给的那么多研究机会。再想想既然中学时能自学大学课程，当下好歹也该试着突破一下困境吧。于是从高中数学课本开始看，一 直看到机器学习专业的教材。然后能看懂论文，了解最新的研究成果。随后自己找了一个把机器学习和图形学结合起来的研究方向——用多个摄像头采集人的动作， 让机器学习这些动作数据，从而能自动合成三维动画。**全力投入一年半之后，我重新自学了数学课，而且在这个方向上发表了10篇论文**。虽然今天完全看不上当时论文的水平了，但是刘老师很高兴邀我二进香港。

在IBM和微软这样的公司实习后，你为什么从此开始了互联网之路？

香港的经历勾起了我对数学和机器学习的兴趣。于是我主动推迟一年毕业，学习机器学习的主要应用——数据挖掘。数据挖掘得有数据；IBM是老牌大厂， 数据应该积累丰厚，于是我投奔了清华的一位师姐刘世霞，去做实习。在IBM发表的论文，将就能让我毕业后腆着脸去敲Google的门。

但是真正值得挖掘的数据不在IBM和微软这样的软件和咨询公司，而是在互联网行业。让我意识到这一点的是我的一位师兄郭奇。他对学术研究兴趣不大，在搜狗兼职，而且兼得很凶——是搜狗输入法的首创者，也是当时搜狗搜索引擎工程架构的负责人。**“输入法的语言模型训练不就是从大家的输入中总结人类语言的规律吗？”**这句话引导了我后来的工作方向——从大众行为数据中归纳人类智能。在郭奇启发了我6年之后，出现了一个热词“大数据”。

大数据时代的理想主义和现实主义

“在大数据时代，先得成为出色的工程师，才能成为了不起的研究员。”

在Google的时候，你参与开发了一个分布式机器学习的工具，这个工具获得了APAC创新奖，并部署在多个Google的产品中。你在开发这个产品中的角色是什么？有什么样的收获？

Google在亚太地区就一个研究团队，全职做研究的就两个人——张栋和我。张栋后来去百度，随后创业。他的故事很多人都了解了。当时我们俩各自做 一些研究。最终获奖的是团队成果的集合。这个团队除了我们俩，还有好几位加州大学、清华、北大、MIT、浙大的实习生，以及几位非常出色的Google工 程师。我负责的一项主要工作是主体模型的分布式机器学习技术。这个研究是张栋做起来的。他换了研究方向后，我换了一种技术思路接着做。这一做就是7年，跨 越了我在Google和腾讯的职业历程，也影响到我目前的工作。

要说收获的话，有两点体会： （1）各种互联网服务收集了大量用户行为数据，这些大数据都是长尾分布的；但是研究领域总体仍然专注在基于指数分布构造的机器学习模型。**这样的模型计算方便，但忽视了数据中长尾的部分，也就忽视了大数据中最重要的部分。 （2）每一种有价值的算法，都值得拥有独到的并行计算架构**。做分布式机器学习的人不可迷信特定框架，比如MapReduce、MPI或者Spark，不要试图套用这些架构来描述各种算法，

而要有能力开发自己算法适用的框架。在大数据时代，先得成为出色的工程师，才能成为了不起的研究员。

这些可能都算是比较新鲜的想法，不一定大家都认同，但是没关系，我把我亲身经历的很多大数据研究工作，简要描述在《分布式机器学习的故事》这一系列博客里了。读者自有体会。

在研究之外，我的很多工作在四处出差，把研究成果应用到产品里。为此我拜访过Google分布在全球很多地方的产品团队。俗话说“是骡子是马，拉出来溜溜”。去了之后，先拿产品数据做出实际效果，给产品团队展示之后，才有机会说服他们使用。这样的工作，也是我们获奖的一个原因。

您致力于把纯研究和业界的需求结合起来，这样做的原因是什么？业界缺乏对于这方面的关注吗？

直接原因是我第一份工作在Google，Google是一个工程师文化极强的公司——这里的老牌研究员个个都是顶级工程师。而搜索引擎这样的产品的 用户体验主要是技术水平决定的。Google因为有最强大的并行计算技术，所以能索引全球网页和支持精准匹配，所以用户体验第一。对技术水平的孜孜以求， 不仅弱化了传统产品经理的角色，也模糊了工程师和研究员的界限——**每一个追求技术突破的工程师，都自然会去读论文，追踪技术前沿，也就成了研究员**。那么研究员也就别自高自大地指望自己设计了算法交给工程师去实现了。因为这个原因，Google里虽然有研究员，但是没有研究院，而且研究员的考核成绩与论文专利没有关系，主要看对产品的贡献。

腾讯的情况和Google的比较类似，都是利用一个平台支撑起很多业务，这样产品线都很丰富。Google的搜索技术在Gmail、Map、 Youtube、Google Now等产品里都有体现；腾讯用QQ汇聚用户，支持了网站、游戏、社交、无线等很多业务。丰富的产品线会收集到海量的用户行为；而数据挖掘研究的目标就是 从数据中归纳用户行为模式，让产品体验更便捷。但是处理大数据对硬件和软件都有需求——**大部分大学和研究院没有足够的计算资源来处理大数据；并且很多研究员也并不擅长设计适合于自己研发的算法的并行计算架构**，于是往往套用现有架构，容易造成系统性能、容错或者可扩展性方面的限制。

为了帮助学术界和工业界融合，我一方面分享我自己做大规模机器学习的经验，抛砖引玉；也欢迎教授来公司做访问研究，同学们来做实习；同时也和腾讯的 同事们一起为国际数据挖掘大赛出题，比如KDD Cup 2012和ICME Grand Challenge 2014——这些比赛题目都是基于真实的业界数据和真实的业界问题。希望能帮助学界了解业界。互联网行业里有一些学界业界交融好榜样，比如卡内基梅隆大学 的Alexander Smola教授。他在Yahoo!有丰富的业界经验。在成为Principle Scientist之后，去卡内基梅隆任教，传授业界积累的大数据挖掘知识。同时在Google做访问研究，保持研究水平的领先。我相信将来会有更多研究 人员像这样两条胳膊都撸起袖子。

离开Google加入腾讯的原因什么?

这里原因很多。总体来说和Google在中国的业务发展势头不强有关。另外，作为一个“土博士”，我要是不了解中国互联网行业，会被人笑话浪费机会 的。而要了解中国互联网，腾讯是最好的学校。这几年我们津津乐道Facebook以及它支持起的Zynga这样的游戏公司。但是在此之前很多年，腾讯就成 功的经营了世界上最大的社交网络QQ，并且依托它成就了网站、游戏、社交等很多业务群。其中社交业务群下的QZone这一个产品拥有的用户数量就和 Facebook同量级。今天，在无线互联网上，微信又在强力支持游戏、移动支付、和O2O。

Google让我见识和实践了世界一流的大数据技术，腾讯给了我了解互联网业务的机会。这是两家很伟大的公司。

在腾讯的时候，你创造了孔雀这个成功的并行机器学习系统。可以向我们介绍一下这个系统吗?

孔雀是一个主题模型的并行训练系统。主题模型是一种机器学习方法，它从文本中归纳“语义”，每个“语义”是一组表达同样意思的词。这个归纳过程通常 比较耗费机器和时间；但是一旦归纳结束，得到了主题模型，那么机器就可以在几毫秒之内理解任何一段文本（搜索词、广告、商品描述或者网页内容）表达的语 义，从而在语义空间里比较用户意图（搜索词）和广告、商品、网页的相关性。而相关性是现代搜索引擎、推荐系统、广告系统的核心要素之一。**主体模型除了用在文本数据上，也可以用在用户行为数据上——此时它就是一个先进的协同过滤推荐系统。**

孔雀系统应用在搜索广告、情境广告和目前比较火的广点通系统里。在前两个产品中，孔雀被用于分析文本数据，归纳自然语言的语义，从而更好地匹配 query和广告，以及页面内容和广告。在广点通中，孔雀被用户理解用户行为数据，从中归纳用户兴趣，从而计算广告和用户兴趣的相关性。

学术界对主题模型的研究从1990年开始。目前可以从数百万文本中归纳数千语义。但是2006年开始，Google的Rephil系统就可以从好几 个数量级大的文本数据中归纳几十万语义，从而奠定了Google AdSense广告系统的相关性的基石，最终帮助AdSense成为Google收入的半壁江山。

据说对于这个系统，你酝酿了好多年，是什么样的起因？这中间的设计有过什么样的变化？

最开始我对Rephil很感兴趣，但是因为种种原因，阴差阳错得没能加入那个团队。**同时我感觉自己一直在跟进的一个学术研究方向，有可能发展出一套新的，规模甚于Rephil的主体模型系统**。所以渐渐有“彼可取而代也”的想法。只是验证这个想法，用了四年时间，分成了几个阶段。

最初的想法在Google工作时形成。走的时候，学术界正在研究很火的LDA模型（和Rephil的模型不同）；并行化方法是MapReduce， 这是Google里最有名的并行化框架。后来发下MapReduce在计算任务安排和分布式文件系统I/O上耗时比实际计算可能还要多，于是尝试使用传统 的MPI。但是MPI不能很好地支持自动错误恢复。于是又改用Google Pregel。Pregel基于一种称为BSP的并行化思路，几乎和MPI一样久远。BSP虽然考虑了容错，但是容错需要cache所有进程的通信记录， 往往导致内存不够。从这时起，我渐渐意识到通用的并行计算框架，很难满足主题模型的需要。同时，我也注意到Google里很多成功的大规模机器学习系统都 用自己独特的并行计算框架。

我到腾讯工作之后，仍然想继续这样的研究。但是当时刚去的时候就一个人，而腾讯的搜索和广告业务在初起阶段，一时之间很难有对基础研究的大规模投 入。于是我放下了这个研究想法，和志同道合的同事们一起专心做了两年多广告业务。直到业务趋于稳定，我自己也被提升为负责广告算法和策略的技术总监之后， 才又开始尝试。

2012年的国庆假期，我用Go语言写了一个尝试性版本。叫尝试性，是因为这次我换了模型。新的模型基于一种叫hierarchical Dirichlet process（HDP）的数学方法。有此考虑是因为之前的研究经验让我意识到LDA和其他很多主体模型（包括pLSA、RBM、NMF、SVD等）都不 能描述长尾数据，而是专注于从高频数据中归纳语义，得到的自然是“主流”语义。**可是互联网的精髓在于服务用户的“长尾”需求**。LDA 即便并行化做得很成功，能归纳很多语义，但是去掉重复语义之后，结果往往就几百个主流语义。 Rephil的模型可以近似描述长尾，但是复制就成了抄袭。HDP也能描述长尾，但是并行计算非常复杂，很难通过减少进程间的交互，切断数据依赖。而切断 数据依赖是大规模并行计算的基础。所以我们对HDP做了修改，让它计算起来像LDA那么简单。此外，对并行计算方法也做了很大的改进。**Go语言的开发效率比C++和Java都高很多，让我能在七天假期里尝试新模型和新的并行化方法**。尝试结果从实验效果上看很有潜力，于是我决定开始真正开发一个大规模系统。

当时我负责的团队有几十位工程师，能使用近千台服务器，而且我的上级领导对这个项目非常支持。但是考虑到我们团队要承担KPI压力，所以实际上除了 我，只有另外两位很年轻的工程师（赵学敏和孙振龙）志愿全职投入，不到团队总人数的十分之一。当时每天的工作时间主要花在团队管理上，编程的时间都在八点 钟下班之后。这样的经历持续了大半年，直到在400台计算机上的并行训练试验效果初见成效后，有更多同事热情兼职加入。大家优化了训练系统，也做了很多将 研究工作应用于实际业务的工作。

做完这个项目你有什么体会?

从这段经历可见大数据技术推进的不易——数学模型的改变和并行计算方法改进密切相关，没法拆开，一部分给学术 界，一部分给工业界。而学术界距离业务比较远，不容易接触到真实的大数据，也很难找到数百台机器，只能在工业界做。而在工业界势必要平衡业务压力和基础研 究两者之间资源分配。先进的研究往往尚未被大部分人理解，又需要比较长的时间——要在此期间获得团队（包括研究、工程、产品、销售）的理解和支持，是对项目主持者的全面考验，需要主持者之前的声誉和撸起袖子来的实干精神，给团队注入信心。这其中领导的支持当然也非常重要。一年前我们的团队经历了一次重组， 新任领导也是搜索和广告业务的行家，对我们的技术研究仍然非常支持。

春节前有一次聚会，席间我回顾了这段经历。在座的百度的余凯老师表示理解，总结说：“今日中国是极端的理想主义和极端的现实主义的结合”。我甚感共鸣，其实古往今来莫非如此。

孔雀之外，把研究成果和业界需求结合后的产品有哪些?

因为我一直在公司里工作，研究都是冲着产品和实用做的。在Google工作的时候，除了自己努力往Google Orkut、生活搜索、音乐搜索等产品里推广，同时也有其他产品（Reader、News）的同事主动尝试。此外，有些结果在开源届有应用。比如北大的实 习生李浩源主导的一个分布式频繁项目挖掘的工作，后来被Apache Mahout系统采用。我和工程师白红杰开源的pLDA项目也有很多用户。后来几位实习生同学进一步改进，成了pLDA+。pLDA和pLDA+一共被 200多项后来的工作引用。

在腾讯的时候，你发明了很多新的机器学习和数据挖掘的算法，这些有没有反过来在学术研究界产生贡献?

在腾讯的工作和业界结合更紧密。因为工作比较忙，没有时间仔细写论文，所以暂时没有论文发表。但是有一些学术会议上的分享和几个开源软件，包括我用 C++写的一个MapReduce的实现http://code.google.com/mapreduce-lite，以及在此基础上开发的一个并行 logistic regression 训练系统 http://github.com/wangkuiyi/lasso。

成长的秘密

“因为大家都敢于创业，所以避免了寡头垄断；而不是因为没有寡头垄断，所以更容易创业成功。”

国内大型互联网企业（如腾讯）和硅谷的互联网企业在管理机制上有什么大差别?

现代互联网企业之间有很多交流和学习，在管理经验和机制上其实大同小异。当年Google中国的同事在能力上和美国的同事没有什么区别；腾讯同事的技术水平也和LinkedIn的同事没有什么区别。**如果说有差异，更多是在文化上的**。

在国内，人多资源少。几百年来中国的文化就强调竞争，流传下来的口号也很多，比如“吃得苦中苦，方为人上人”。为什么要做“人上人”呢？很大程度上 是为了多吃多占吧。当代的中国孩子也是从小就被迫和邻居家孩子比成绩。长大之后，习惯性地和同事比较年终奖、晋级晋等。可是太过计较小节，就容易忽视了大 方向。而且自己人之间的恶性竞争，削弱民族凝聚力。

资源竞争也体现在高额的房价上。房价束缚了很多年轻人——为了攒钱首付，为了能稳定的还贷，在工作中谨小慎微，不敢直言直谏，接受很多苟且和无奈，放弃了成就业务和完备自己的机会。这和二战后的日本以及我出差时见到的今日印度很像。

在美国，地多人少。即使在硅谷，由于最近几年大量中国和印度移民涌入，房价提升。但是换算到每平米单价，仍然在北京上海广州深圳之下很多。再加上平 均工资水平相对较高，在这里留学和工作的年轻人买房时是不需要老父母帮助凑首付的。在国内普遍三十年还贷，在这里一般是十年之内。

在资源竞争相对宽松的环境下，西方的教育也相对宽松。强调人格的培养，而不是知识的灌输。中国有句古话“宰相肚里能撑船”，就是有多大胸怀做多大事 业的意思。相对宽松的环境，给人更大的挥洒空间，从而不计小节，就像平生慷慨的班都护和万里间关的马伏波——他们自己以及我们这些后人恐怕都不在意他们是 行政干部几级。我理解这是硅谷里很多人有更大的胆魄创业的重要原因——**因为大家都敢于创业，所以避免了寡头垄断；而不是因为没有寡头垄断，所以更容易创业成功**。

从什么时候开始从事管理工作？更喜欢纯技术工作还是更倾向于技术管理工作？

我更愿意做我喜欢做的事。如果这件事需要做技术，那就做技术；如果需要做管理，就做管理；如果需要二者兼顾，那么就累一点儿，奋力兼顾一下。

在国内的科技行业，尤其是大公司，有一种说法“三十岁之后就干不动技术了，要尽早转管理”。其实我也见过很多很早转管理，技术上不再长进，丧失了技术行业的核心竞争力，从而不得不留在大公司混派系的例子。

另一方面，我的同龄人里有很多很出色的榜样。比如最近在硅谷认识的一位朋友杨文杰——上海交大数学系本科。读书期间就创业。毕业后，为了进一步开拓 视野，先后在J.P. Morgan（香港）和Summit Partners（美国）工作。一边工作一边在斯坦福读MBA以了解美国环境。**做了多年准备后，现在又离开大公司在硅谷创业，用人工智能技术支持商业拓展**。他这样在数学、计算机、金融、投资、管理等多个方面努力学习、融会贯通，功底是和扎实的。而为了做到这些，他每天的体能训练也很扎实。并且为了保持精神状态，每天洗冷水澡。

另一位在Google认识的大哥王欣宇，在总结自己的职业发展时有一个四字口诀“募投管退”——他选择的职业发展路线，使得他在募集资金、选择投资业务、管理团队、和公司上市四个方面都有锻炼。

我见到不少朋友给自己打个标签：技术人员或者管理人员，甚至区分工程师和研究员。其实这些标签往往是一种束缚，而人的本性其实是追寻自由成长的空间的。

在编程语言上，你的选择是什么？都应用在了哪些项目上？

我接触过的编程语言不少，因为对语言很好奇。随意数数至少包括BASIC、LOGO、6502汇编、Fortran、Pascal、C、C++、 80386保护模式汇编、COBOL、Tcl、Awk、Javascript、C#、LaTeX、Maxima、Maya、Emacs Lisp、Scheme、Common Lisp、Erlang、Radeon GPU汇编、Cg、Java、Python、Haskell、Objective-C、Go。其中对我影响最大的是Lisp，是我的同学王垠教我的，让我 接触了一点计算的数学本质。在微软图形学组实习时学了GPU的汇编语言，后来用GPU写并行机器学习算法的时候用过Cg。我工作中用的语言主要是C++。 从写Peacock开始用Go。简单的分布式数据处理用bash+ssh+awk代替MapReduce。

你一路走来加入的都是一些业界闪耀的公司比如IBM，Google，腾讯，以及现在LinkedIn，这是你刻意的选择吗？你会给计算机相关专业的大学毕业生什么样的择业建议？

我很珍惜我的同事们，他们给了我很多帮助和提示；但没有刻意选择加入大牌公司。毕竟如吴军在《浪潮之巅》里说的，闪耀的牌子都在一波波的浪潮中过去了——今天毕业入行的人记得Sun的不多了，知道DEC的基本没有。

我在择业时也有很多茫然不决的时候。但是我有个好榜样，是原来Hulu.com的engineering VP张小沛。她对择业的建议很简练：“最重要的是知道自己要的是什么”。

原文链接：http://www.lewuxian.com/?p=211919

即使别人是码农，你却不该是

本文来自于：http://blog.sae.sina.com.cn/archives/3548

好几天前，在微信里，有个童鞋给我留了这么一段话：

「程序君，昨日知乎日报上出现的那篇《为啥中国的程序员都被称为码农》（以下简称「码农」），看完实在心酸，作为一名还在大学校园即将走向“码农”大军的愣头青，想请教您，你对那篇文章有啥看法？上面的说法属实吗？中国程序员的现状大体是怎样？麻烦指点」

我大概看了一下那篇文章，说的有些道理。但程序君认为：别人是不是码农与你无关，你不该成为那篇文章作者眼中的码农。作者说码农一词强调程序员「地位低下、枯燥和劳累」。作为一个程序员，我也来随便说说。

收入和地位

一般而言，程序员的收入水平不低。我没有具体的数据，但在一线城市，程序员的平均收入应该都能达到该市的中上水平 —— 我猜top 30%左右。2012年，我们在校园招聘的时候，很多面试后非常心仪的同学（清华，北大，中科院等）最终都拿着十几到二十万的薪资去了B，T等公司，有一 个我们特别中意的iOS工程师，被我们追了很久，但后来最终还是被某著名游戏公司招安，拿了二十好几万的薪水 —— 这可是程序君工作了好几年后才能拿到的package啊。

所以你说程序员的收入低么？为什么你的收入会低？为什么你怕你未来的收入会低?

程序君有个朋友，也是途客圈的前员工，本科来实习前已经有很多独立的项目经验，掌握了python/django和iOS的开发能力。他聪明好学， 上手能力非常快，稍加指点就能从事重要功能的开发，勤奋程度又不输于程序君，所以进步神速。后来途客圈的很多核心功能都交由他来负责，某些功能程序君都自 认为无法做得比他好。一年半后他从途客圈「毕业」时，已经是各大公司争相想招致麾下的「面霸」，同时拿到了好几份offer，最终去了某搜索公司，现在前 途一片光明。

我想这是一个很有借鉴意义的例子，尤其对于在校学生。就像之前的『软件开发升级打怪之路』所讲的那样，我们身边有那么多很有意思的问题可以通

过软件来解决，你愿意放弃一部分打dota的时间和精力去解决么？你愿意在解决的过程中排除万难，啃下一个个硬骨头么？

如果你在学生时代就有很多拿得出手的项目，那么在现在互联网热火朝天，人才缺口很大的时代，找一份薪水不低的工作还是难事么？

程序员可能是世界上唯一一份不用太靠学历，不用太靠爹娘，甚至都不用太靠熬日子出头的工种。有人杜撰了这么个故事：

说Python之父Guido van Rossum有天跑去google面试，说了三个词：”I wrote python”，就被录用了。

当然像Guido这样的大牛犯不着主动去找工作的，就像球场上的超级明星，给猎头打个电话，说我想挪个窝了，工作机会就会像雪片一样飞来。这个故事虽为杜撰，但不失一个很好的例子说明程序员「不靠天，不靠地，就靠自己一双手」的本质。你的薪水取决于你能做出什么来。

至于地位，我觉得除了权贵阶层，其他人的地位都差不多。如果不做公务员，没事别老琢磨地位，那玩意说来就来，说走就走。我倒觉得程序员应该多提高自己的品味 —— 至少多学学打扮自己，别拿着中产的收入过得像无产阶级。

另外，建议妹子们也多关注关注程序员这个群体 —— 毕竟能够改变世界的几类人中，程序员算是最好接近，在比较年轻的时候就能看出潜力，也最好玩弄于股掌中的。^_^

工作枯燥

工作枯燥这事真心和你自己的感受有关。首先不是所有人都适合做程序员的，如果你换了不少团队或公司，做什么都觉得枯燥，自己又没兴趣做pet project，那你要好好考虑下自己是否适合这条路。否则走下去，就真成了「码农」一文中的码农了。

有人曾经给我留言说自己不想做业务相关的事，没意思，想做「真正的程序员」做的事情。拜托，我们做的是产品，哪个产品不是和业务相关的呢？脱离了业务的软件，要么是纯粹个人爱好，要么只能在象牙塔里生存。

有人说工作特么没劲，每天干的都是琐碎边缘的活儿，枯燥死了。好吧，你以为程序君做得总是高大上的事情么？程序君最近两周干的活也琐碎得要命，其中 一个任务类似于「从linux kernel的源代码里，把所有.c引用的.h文件摘一摘，只留下真正有用的（但系统还能正常编译运行）」。

工作中这样的活不少，枯燥是有点枯燥，遇到了与其怨天尤人，不如想办法快点将它完成。程序君花了大半天时间，在走了很多弯路写了两个程序后，终于找 到一个巧妙的办法，仅仅写了五十多行python代码就将其完成。效果从最初的方案减少了25%的.h文件一直到减少了95%的.h文件。

我比较不理解有程序员说自己总不得不做重复劳动，所以感觉工作异常枯燥。想想「程序员」这顶帽子带在头上意味着什么？它意味着全世界任何群体都有理 由说自己的劳动是重复劳动，唯独程序员这个群体不能。为何？程序员坚守的信条是DRY（Don’t Repeat Yourself），一件事当你发现你需要重复第二次时，就要考虑将其自动化。做不到这一点的请努力，因为这决定了你的效率和效能。

还举我自己的例子吧。前些日子我要测试几个开发环境，流程大概是下载代码，编译，运行UT。因为开发环境有点问题，所以在下载完代码后我需要对代码 打个patch。这活第一遍我是手工做的，为了验证整个流程的正确性，调整patch等等。第二遍以后我就写了个脚本将其自动化。虽然在我写这个脚本的时 间里，我完全可以对所有的开发环境都一一验证，但脚本化的好处是，我可以让别人用这个脚本也进行独立验证，我也可以在今后几天的工作中反复使用。

枯燥是你看待任务的主观情绪。很多看起来外表光鲜的互联网公司或者软件公司，真正分到你手上的任务就不见得光鲜靓丽。大数据？那是对外美好的商业表 述。你真正做的事情也许是对海量日志进行或手动或半自动分析，枯燥不？操作系统？好吧，你去了以后发现主要做的是本地化，枯燥不？虚拟化？好吧，那里很大 的团队在做驱动开发，枯燥不？

没那么多枯燥。软件就是一个个实现起来非常枯燥的功能有机地组合在一起，为用户（客户）提供价值。无法认清这一点，总认为自己干的就是最枯燥的，那你只能继续枯燥下去，也只能成为「码农」作者眼中的码农。

辛苦劳累

辛苦劳累倒是真的。不过要看你怎么个辛苦法。

如果你在一家各种限制你自由发挥，还以你工作时长为工作态度和工作能力的评定标准，那么，除非你有其它想法，否则应该选择离开。记得我毕业后工作的 第一家公司，有天晚上吃饭，老板问我对team里两个女孩有什么评价，我说她们工作得挺好，合作愉快啊（潜台词是男女搭配，干活不累

^_^）。老板努了努 嘴，说：可她们一下班就回家，工作态度不积极啊。我听着不是滋味，心里就萌生了离开的念头。

程序员的工作绝对不应该用工作时间，是否加班来衡量。如果你的老板给你的评定是「该员工工作积极努力，主动加班，blablabla」，你还愿意这么呆着，那你就别抱怨辛苦劳累。

不过现状的确是是很多程序员都在加班，包括我在内。

有些人加班是真忙。但其实有很多行业比程序员忙得多，比如四大所在的会计（审计）行业，比如投行，咨询。

也有些人加班是刷存在感。

但更多的人加班是为了有一个清静的环境，能做点什么。

要说辛苦劳累，我觉得一个很重要的原因是：这个工种需要你不断更新夯实自己的技能。

如果被迫接受，那身心俱疲；如果主动出击，身体累了点，心灵上的成就感还是不小的。

jQuery之父：每天写代码

作者：John Resig

jQuery之父，同时也是Pro Javascript Techniques和Secrets of the JavaScript Ninja的作者。他目前主持 Khan Academy 的开发工

本文链接：http://segmentfault.com/a/1190000000469890

去年秋天我的支线代码项目 遇到了一些问题，项目进展不足，而且我没法找到一个完成更多代码的方法（在不影响我在Khan Academy方面的工作的前提下）。

我主要在周末进行我的支线，当然有时候也在晚上进行。这个方法对我而言效果不佳。我的压力太重了，我需要在周末努力完成尽可能多的工作（如果没做 到，我会为此感到挫败）。还有一个问题是我无法保证每个周末都有空，而且我也不想把周末所有的时间都花在编程上（失去一切放松娱乐的机会）。

此外，每隔一周进行编码的话，间隙太长了。太容易忘记你正在做什么，或者你还需要做什么了（即使你有笔记）。如果你错过了一个周末的话，问题就更严重了，间隔一下子变成两周了。多周的上下文切换可能是致命的（我有很多支线项目因为这类注意力缺乏而死亡了）。

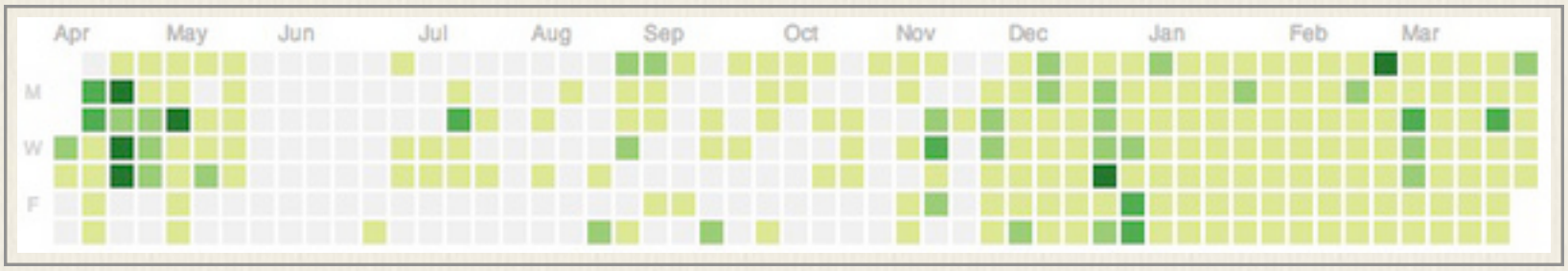
Jennifer Dewalt 去年通过在 180 天创建 180 个网站的方式来自学编程，她的做法启发了我。我决定采用一个简单的策略：每天编码。

我决定为自己定下一些规则：

1. 每天必须写代码。我可以写文档、写博客或者写其他东西，但是这些不能代替写代码。
2. 代码必须是实际起作用的。调整缩进、重新排版不算。如果有可能，重构也不算。（可以进行这些事情，但这些不能是一天唯一的编码。）
3. 所有代码必须在午夜之前完成。
4. 代码必须是开源的，放在Github上。

有些规则比较武断。比如代码其实不用在午夜前写完的。但我担心熬夜导致代码质量下降。同样，代码也不用开源，或者放在GitHub上。我这么做是想强迫自己写代码的时候上点心（考虑可读性，同时较早地考虑模块化）。

到目前为止这个策略很有效。我基本保持了20周的连续工作。我之所以写这篇博客，正是因为它完全改变了我编写代码的方式，更重要的是影响了我生活和心智。



这个习惯的改变产生了一些有趣的结果：

最小可行的编码。 我强迫自己每天花不少于半个小时来写代码（如果少于这个时间就很难写出有价值的代码了，特别是回忆前一天写了什么还要花一点时间）。工作日的时候我有时写得多一点（一般不超过一个小时），周末我有时整天写代码。

写代码成为习惯。 值得一提的是我并不是特别在乎上面的Github图形。我觉得这是这个实验最值得借鉴的一点：这是你为自己做的一个改变，而不是为了满足别人对你工作的评价。节食和锻炼也是一个道理：如果你不在乎提升自己，那么你永远都不会取得成功。

与焦虑作斗争。 在开始这个实验之前，我时常为没有完成足够的工作或取得足够进展而感到焦虑（工作和进展都难以量化，因为我的支线项目没有死限）。我意识到，感觉到进展和 实际推进工作同样重要。这令我大开眼界。一旦我每天持续地推进项目，我的焦虑就开始消散了。我对自己的工作量很心安，我再也没有那种难以承受的渴望，想要 疾风骤雨式地推进项目的渴望。

周末。 以前，在周末完成工作绝对是前进的关键动力（因为通常而言这是我唯一大量推进支线项目工作的机会。）现在情况不一样了——这是件好

事。期望在一个周末完成 一周的所有工作只会让我失望。我极少很完成工作，因此为了完成更多工作而拒绝了其他喜欢的周末活动（例如吃 dim sum，参观博物馆，去公园游玩，陪伴我的伴侣等）。我深深地感到，虽然支线项目是非常重要的，但是它们不应该是生活的全部。

后台处理。 每天编写支线项目代码的一个有趣的副作用是你当前的任务时常在你大脑的后台运行。因此当我去散步或沐浴的时候，或者进行其他不费脑的活动的时候，我在思考 我接下来将做哪些编码，寻找解决问题的好方法。我以前一周或两周编码一次的时候可不是这样。当时时间被花费在思考其他一些事情上，通常是在为没法完成任何 支线项目而感到焦虑。

上下文切换。 拾起支线项目工作的时候总会有上下文开销的。很不幸，重拾整整一周前的思考是极其困难的。每天做一点对此很有帮助，因为间隔时间大大缩短了，让我很容易想起在做什么。

平衡工作。 这个改变最重要的方面之一是我已经简单地学会了如何更好地平衡工作、生活和支线项目。我知道自己每天都要做支线项目，因此我需要更好地管理时间。如果我计 划晚上出去，并且很晚才能回家，那么我需要在早上为支线项目工作，在开始为我主业Khan Academy工作之前。同样的，如果我在外面，而我还没有完成我的工作，时间已经很晚了，那么我需要尽快赶回家去完成工作（以免错过一天）。我需要指出 的是我发现自己把更少的时间花在爱好上了（例如木版画），但是这是一个合理的代价，我得接受这个。

对外沟通。 跟外界沟通自己的新习惯是很有好处的。我的伴侣理解每都必须完成工作，因此有活动安排有时需要据此作出调整。这样我就可以很方便地说：“是的，我们可以出去/看个电影/等等。但是我之后需要写代码。”我的伴侣会理解我，并在安排活动时考虑到这点。

我写了多少代码? 我很难相信自己在过去的几个月写了这么多的代码。我新建了几个网站，重写了一些框架，并创建了大量node模块。我写了如此之多的代码以至于我有时我都忘记自己做了写什么——几周前的工作感觉是很久远的记忆。我非常非常高兴自己能写这么多代码。

我认为这个习惯的改变是一个巨大的成功。我希望自己能持续保持这个习惯。同时，我尽力向其他希望能完成大量的支线项目的人推荐这个策略。告诉我这个策略对你而言是有效还是无效。我很希望能从你那里听到一些有意思的东西。

GDC 2014 之后，游戏引擎市场会有怎样的变化?

作者：@康托耶夫II
自诩是国内最猛的虚拟现实开发者.....好吧 那就之一好了。。。

游戏引擎市场近几年的变化是比较大的 显著的特征是手持端的兴起对引擎的冲击，早前的游戏引擎真正市场化的是unreal,走的商业模式针对大是型游戏公司的，即大公司购买授权号（巨贵 几十万刀吧）进行开发，买不起的自己玩去。

对于小型的游戏公司，会采取低价策略（几万刀吧）加上分成协议，也就是引擎商根据你们游戏的销量抽成，那样万一有一匹黑马公司突然上位，游戏引擎开发商 （主要也就是Epic) 跟着捞一票。

那微型公司或者个人开发者呢，一般就用不起unreal这类的高贵引擎，会考虑Ogre或者自己就是技术大牛便自己DIY开发一款（游戏程序员以自己开发引擎为荣是很长时间以来一直有的风气,那开发的如何另论吧，有激情总是好的么）

不过，不过之前的话大家都可以当空气对不对。。。

不过后来出现了搅局者， 这个搅局者就是大家众所周知的unity3d, 没有它，就没有现在的udk,或者cryengine 的个人版本，

据说故事是这样的，有几个北欧乡下的吊丝青年，也就是上面说的小公司或者说indie开发者，不堪高大上游戏引擎商的无视和凌辱，决定为了第三世界的游戏 开发者们为了亚非拉的屌丝游戏人们站出来，开发一款英特纳熊耐儿的游戏引擎，让全世界的无产者们联合起来,过上性福的日子。

当然事后怎么吹随便他们，反正他们现在一片光明，牛的不得了，but,当时未必真是这样，unity早期版本是仅有mac的，而且看着也特别怪异，主要应用是拿来做虚拟现实，那mac下面的虚拟现实有个毛毛用啊，所以也没多少人用起。不过早期由于在mac下面的软件都充满了自己是设计师的优越感的自觉，在 界面设计上unity3d不知是否受到此风影响，非常的简洁易用， 这种基因到现在依然存在，用unity开发是非常顺畅少滞涩感的，总觉得那个按键就是应该在这里。

在默默无闻的版本一之后就是业界大事 件， unity2.0的出现，这个奇葩引擎的升级给整个业内带了了巨大的冲击，因为它不仅仅是支持了win mac的跨平台开发(也就是你的工程文件直接拷贝到任何一个平台下双击打开就可继续开发）而且开始支持ios和android的操作系统，这一非常富有远 见的功能整合，为unity的崛起奠定了基础

来说说unity2.0出现之前那些其他的游戏引擎们都在干啥，当时的业内风向，是觉得PC端游戏最高，Cryengine代表业内引擎最高画面质量未来 无限好像樱木花道，Unreal代表次世代游戏工业标准大约就是流川枫，其他游戏引擎如ogre,renderware,gamebryo都是类似樱木三 人组的等等！

Big-
world这类网络功能较好但是画面普通的优秀引擎也是依然要被两大豪门们鄙视的
“天天搞个破网络，画面也不搞搞好，像个什么样子！你说 为啥不好好学习”

再 或者说强劲的in stuido engine如forestbite 或者顽皮狗的引擎 小岛的fox 啥的也就是藏在深闺默默挣钱，毕竟程序员用的引擎和商业化的engine是不一样的，很多厉害的引擎功能再强外人也是用不起来，就是个窝里 横，然后哪些PS 或者XBOX上Wii上的开发也都是各种公司各种玩法，所以商业市场上主要就是流川樱木和等等们，

但是还有水户洋平呢，unity3d是一家眼光非常长远的公司，每次的新功能发布看上去都有些遮遮掩掩的欲语还休，其实目标很明确。unity2.0标榜 自己跨平台，但是其实是对android和ios的手游市场产生了兴趣，当然这些是在他所谓全平台支持的战略下进行，它试图通过一个工程文件，一次性无痛的发布包括ios android xbox wii ps3在内所有的平台，（真正做到是unity3.0时代的事情），但当时已经对所以游戏开发者造成巨大的震撼，大

家都意识到以unity3d这样方便的 开发工具，如果能够一次开发就针对所有平台进行发布，是非常有吸引力的，而且unity的价格仅仅是基本版1w5人民币，针对每个发布平台再加1w5人民 币即可。

同时，提供了独立游戏开发者功能稍微受限制的免费开发版本，一时间所有游戏开发者都把目光投向unity,即使那时候unity的画面和技术都不能和cryengine和Unreal比，但是他易于开发的特性，多平台无缝切换和发布的功能，以及非常廉价的价格，加上公司CEO吹嘘的“民主化的引擎！”造成的社会影响是----

-unity3d 是民主的 美好的 三个代表的， cryenigne和unreal变成了特权阶层和陈腐贵族阶层的代名词，当年的社会舆论一边倒的给unity奉上最美好的褒奖和恭维，unity俨然就 是明日之星，而他后续开发的版本3.0 4.0也不符众望的改进了动画系统和渲染，寻路，场景优化 等对于独立游戏开发者们颇为头痛的问题，继续得到大家的追捧（大家都不想做的事情算是unity帮你开发完了）

回头来看看樱木和流川，突然发现没人来找他们打波了，大家居然都去请水户洋平，开始当然一声冷笑，看他出场费那么低，有什么了不得，但是越往后越觉得不对劲，所有的单子都跑去水户洋平那里了，两位大佬天天坐板凳，其他的等等们更是没了活路，这才发现水户心气平，球技好还收钱少，成了都教授万众瞩目还不骄不馁，俨然是要取而代之的意思，看来以自己的价格，再没人请来打球了。

于是，UDK(unreal development kit)开始出现了，这种表面上娇滴滴的说我们只是给独立开发者玩玩拉的姿态，其实就是考虑自己身价太高，一般人接受不起，索性学习unity战略，走低 价免费授权，抢占市场，心里已然是服了软，crytek很快也采取了类似的策略。 收效。。。 其实勉强， 毕竟是没法子脱了高大上的外套跳进坑里和unity肉搏，赢之也已湿身，输了就没底裤了，反正不好看相。

回到今年，现在的状态是Unreal和Cryengine都闭口不谈unity，假装两贵族决斗，米有unity你个草民的事情，其实心里都想着 unity的手机市场，斜眼看着又拿不到，还不能说破，心痒难耐，但是嘴巴上只是说“喝， 哈哈哈 ， unreal,你这招落的下风， 嘿！ 你cryeninge内力不足了

吧"，Unity就在旁边蹲着嗑瓜子毫无形象，没事过来踢他们屁股一下，只做没看见，强挺贵族风范。

所以今年的状态是unreal和cryeninge都推出了19美金和9美金的包月套餐，来对抗unity5(他们不会说是对抗unity的 嘘） 咳咳 对抗对方。试图在自己几乎完全失去的手机市场站稳脚跟，毕竟未来市场是手持端的。

Unity呢，其实毫不寂寞，所谓樱木流川他都没有看着眼里,野心藏了好多年现在终于有资本干大的，可以图穷匕见了，最终目的是仙道！ 业内小霸王的adobe！ 终于在预谋了多年之后推出了webgl的支持，
html5阵营一直不能完全取代flash的原因就是缺乏优良的编辑器，而且unity终于介入这全新领域，站在一帮早对adobe看不太顺眼的手机大佬鞍前，对flash市场造成严重的威胁，身后是unreal和cryengine抢自己剩下的饭粒，呵呵.....

adobe正在仰望星空 不知道在想啥，也不知道意思到了没

对了 在旁边的角落里面 有个叫coco2d的胖子冷眼吃肉。

以上纯吐槽，同业留命!

原文链接：http://www.zhihu.com/question/23103406/answer/23615860

FEX官网诞生记 + 完整PSD下载

作者：@Rayi-
一个热爱开源和视觉项目的前端工程师

这又是拖延症爆发的结果

本来早就说在网站上线后就写一篇这样的文章的，结果拖延症犯了，一直到现在还没写好。

最近实在是觉得无颜再拖，所以赶紧写出来给大家随便瞅瞅。

请勿拍砖，拍砖请自带砖头！

且先说说这个网站

网站也就是目前百度FEX团队的官网了，之前接到任务后，花了一些时间做了设计图，然后就开始开工去做静态页面了。

不过因为要用Jekyll来搭建，所以又去了解了一下Jekyll的相关信息。至于为什么选择了Jekyll，且听我慢慢道来。

目前FEX的官网使用的是Github Pages的服务，你可以到 这里 来查了解关于更多Github Pages的信息。

网站的设计图也放出来给大家了，如果对网站有兴趣的，可以到我们的网站代码github上去fork一份代码，整个网站都是开源的。

如果你有对代码做了什么修正，也欢迎 pull request 提交代码给我们!

为什么用 Jekyll

首先 Github Pages的服务本身是不能运行类似php这样的动态语言的，因此一般大家都常用的方法是写静态页面。

那么可选的方案一般就是:

1. 自己写静态页面，每次写了之后更新。
2. 通过一些工具来实现静态页面创建。

但同时，我们还有一些其他需求:

1. 文章能用markdown格式来写，不涉及展现的html代码。
2. 整个网站放在github上。

于是结合起来方案就是利用工具来维护，生成静态页面，然后更新到github上。但是这种方案的问题是，每次提交前必须要生成静态页面，而这个动作不太好做成自动进行的（因为我们的文章作者有很多，不一定大家都想的起来做这个动作）。 那自然，我们想去了解是否能提交后生成静态页面呢？？

我们看到Github Pages支持了Jekyll，意味着当你将Jekyll的项目代码提交后，会自动给你生成相应的静态页面。于是我们的问题愉快的被解决了。

这里插播一下，除了Jekyll之外，其实还有一些其他工具方案来生成页面的，大家不妨看以下 Rank 写的这篇 《用 hexo 在 github page 搭建博客》，里面有提到其他的工具方案。

其他杂七杂八的

个性化域名设置，去看github的文档就行，或者直接穿越去看看: https://help.github.com/articles/setting-up-a-custom-domain-with-pages 。

评论系统本来之前用的 Disqus,但是因为毕竟是国外服务，加载速度上有点慢，所以最后改成了多说，速度快多了，而且可以微博登录哦。

其他好像没什么想介绍的了，如果有对其他信息感兴趣的，欢迎留言评论!

对了！我在文章列表那里参考了淘宝UED的文章块样式，这个得写出来，不然就是刺果果的抄袭而且不承认了!

福利包

如之前所说，我把网站设计稿和一些Banner图片的设计稿都扔出来了。

如果，我是说如果你对我们的官网设计感兴趣，那么，你可以移步到 下载PSD文件来下载相应的PSD文件。其中包括首页和文章页的设计稿。

也欢迎看看我之前写的其他文章：《妹纸+基友技术交流会，有图有真相哦！》

原文链接：http://fex.baidu.com/blog/2014/04/about-this-site/

资料推荐

机器学习论文与书籍推荐

《跟我学Shiro》PDF完结版下载

有哪些信息安全方面的经典书籍？(from 知乎)

#百度技术沙龙#第49期讲师程一仕【MooseFS和Redis在海量存储下的架构改进和性能提升】的主题分享PPT

Beej's Guide to Network Programming

斯坦福大学-游戏编程-A*算法

Learn Version Control with Git